caBench-To-Bedside

Design Document
Version 1.0

[*Insert approval date of document*]

Revision History

Date	Version	Description
March 6, 2007	1.0	Draft document
July 29, 2007	1.0	Updated for beta release

Index

List of Figures

1 Introduction

This document explains the design of the components and modules present in caBench-To-Bedside (caB2B) project. It provides details of different components that are being developed as a part of caB2B application and may be shared across other applications.

2 High Level Architecture

2.1 Overview

This section describes the overall architecture and high level design of the caB2B.

The caB2B application is a highly user interaction-rich application that will allow the user to perform the following:

- Search and query different grid enabled data services to acquire data sets of interest
- Save data sets and create an 'experiment' in order to analyze and visualize this information
- Perform different analyses using different grid enabled analytical services
- Visualize analysis results using a rich collection of windows
- Execute workflow jobs, time-taking queries, or analyses asynchronously
- Share experimental result amongst multiple caB2B users

The caB2B application has a client-server based architecture.

The caB2B client is a desktop application (implemented in Java Swing) which provides the user a graphical user interface to search for data sets of interest, create experiments, and view different analysis results.

The caB2B server performs backend activities associated with user interactions. The server caches static data like classes and attributes from domain models and their associations as well as query execution results. Following diagram shows overall architecture of caB2B. We will see the components shown in this diagram in later sections

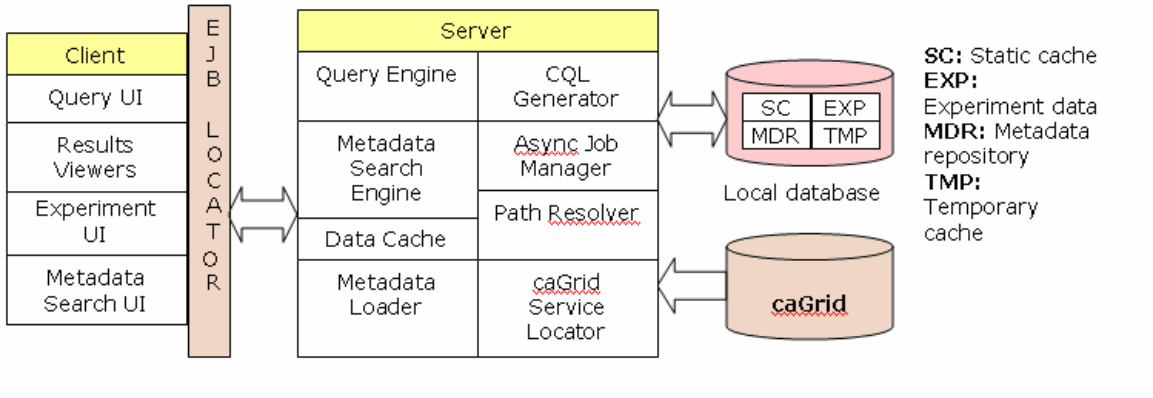

Figure 1 caB2B Client-Server Architecture

2.2 Why caB2B uses client-Server based architecture?

The rationale for selecting a client-server based architecture is as follows:

1. A centralized caB2B server avoids the need to install a database per client.
2. Server stores common data required by all the caB2B clients which includes
    a. The parsed UML model classes and attributes and their associations obtained by downloading models registered in the caDSR repository.
    b. All possible paths between pairs of UML classes.

3. Disk space consumption is reduced on the client as common data is stored on the server.
4. Common data which needs to be refreshed periodically from some source external to caB2B like downloading UML models are fetched by the caB2B server. Thus every client does not need to acquire such updates as this activity is centralized with clients connecting to this sever to receive the latest updates.
5. The caB2B server caches static data like all the classes from the domain models and associated paths resulting in significant performance gains.
6. Asynchronous tasks such as executing analytical services, executing complex queries and workflow management may be performed by the caB2B sever allowing the caB2B client to be interactive. The user can perform other tasks until the caB2B server completes its task and returns results back to the client.
7. User created experiments and query results are stored on the server. Hence these results and experiments may be shared across multiple users connecting to the same caB2B server.

2.3 Client-Server Communication

Communication between the caB2B client and caB2B server occurs through RMI-IIOP i.e. "Remote Method Invocation over the Internet Inter-ORB Protocol". EJB is a part of Java RMI-IIOP i.e. EJB is a remote object and is callable from a different JVM. For more details on this use the references section. The diagram below shows the architecture of the caB2B application portraying how the client interacts with the server using EJBs:

2.3.1 The reasons for using EJBs

- EJB allows the client to have a remote Java object easily (i.e. the stubs are generated automatically by the container).
- It is very easy to call EJB from a standalone client. With an EJB lookup and creation logic are encapsulated in one place. The client code is not aware that there is an EJB on some remote machine which is catering its request. The client just calls methods as if they are being called locally.
- An EJB's life cycle are managed by a J2EE-compliant server.
- EJB provides failover and load balancing i.e. one instance of a stateless EJB can cater to more than one client simultaneously.
- An EJB can "publish" a Java API centrally as a RemoteInterface. Such an API is referred to as a BusinessInterface in caB2B. The class providing that API may be looked up and methods may be called from any remote web application/standalone application.
- All EJBs are stateless session beans. For example the query engine related EJB executes the user specified query and returns the result back.
- EJB is an open standard designed for vendor independence. The EJB specification is developed and supported by all major open source and commercial vendors in the enterprise Java community.

Dependency on EJB:
One important point to note here is that none of the business logic components have any dependency on EJB. In fact, components like query interface, metadata repository, metadata search and diagrammatic query view (DAG) are some of the components that are reused across caTissue Suite and caB2B. Note that caTissue Suite is a web-based application developed in Java Struts framework whereas caB2B is a desktop application developed in Java Swing framework. In spite of these fundamental differences, reuse of most of the

components illustrate that the business logic components does not have any dependency on the technology used to communicate between client and server (EJB in this case).
Note: We are currently using EJB 2.1 and will be migrating to EJB 3 in the next release of the caB2B application.

2.3.2 A Sample scenario

During server startup each EJB's Home Object (i.e. factory for creating EJB instances) is tied with a name in the JNDI (see references) tree on the same server. When the client needs to call a method on the server, it does the following:

1. It asks the EJB-locator locate the appropriate EJB instance (in the form of a **BusinessInterface**) that provides the required functionality. Each locator instance is aware of which JNDI tree to refer.
2. The Locator looks up the 'Home Object' of the corresponding EJB in the JNDI tree and uses it to get the EJB instance.
3. The client calls the required method on this business interface.

The following sequence diagram describes a sample flow of EJB lookup remotely. The example is that of a finding all paths between two UML classes, also referred to as entities in the application:

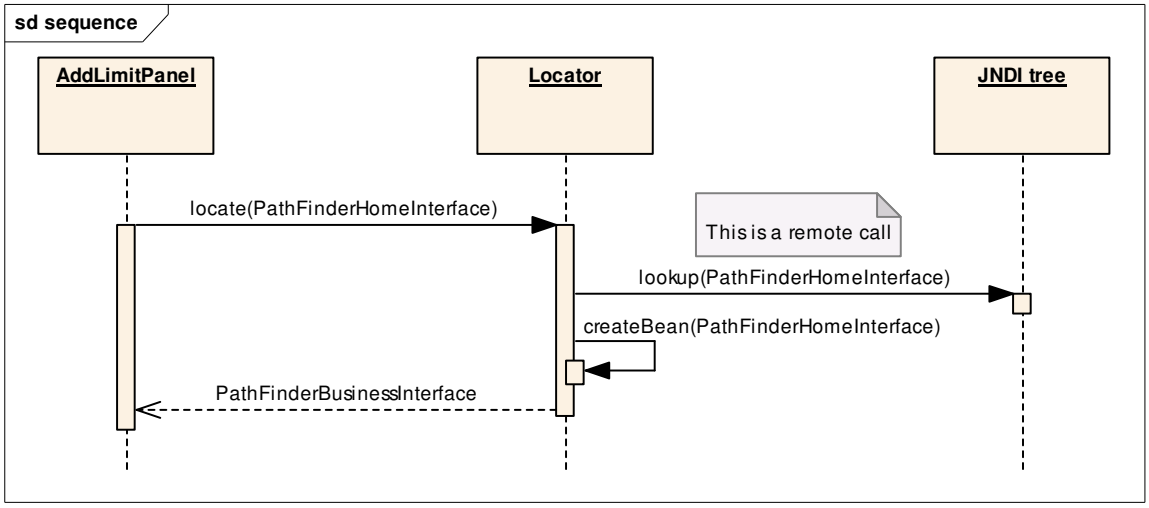

Figure 2 Example of client server communication via an EJB lookup

1. PathFinderBusinessInterface has method getAllPossiblePaths(). It accepts source, destination and returns a list of Paths.
2. An EJB, PathFinderBean implements this interface.
3. Its home interface is PathFinderHomeInterface. EJB's remote interface i.e. PathFinderRemoteInteface will extend PathFinderBusinessInterface.
4. The UI will call Locator to find the appropriate class for finding paths. Locator will lookup the PathFinderHomeInterface from the JNDI tree and will call create () on it which returns PathFinderBusinessInterface. Locator will return that to the UI.
5. The UI will call getAllPossiblePaths() on PathFinderBusinessInterface to get the list of Paths.

2.3.3 Classes involved in client-server communication

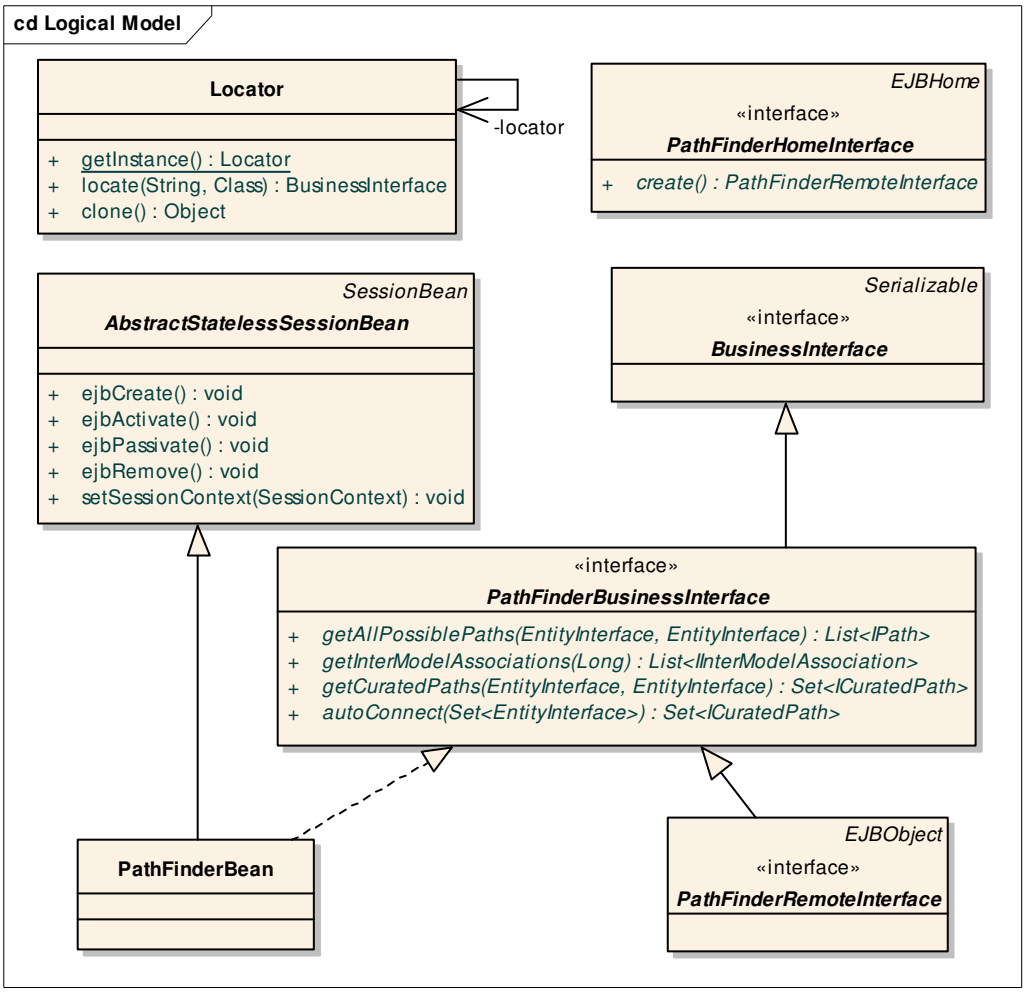

Figure 3 Class diagram showing usage of EJB with PathFinderBean example

Locator is responsible for all EJB lookups. This is a singleton class. The caB2B server to be contacted in configured in "cab2b.properties". Locator looks up the Home Object of corresponding EJB in JNDI tree and uses it to get EJB instance. It has the following method to lookup **BusinessInterface locate(String ejbName, Class homeClassForEJB)**

AbstractStatelessSessionBean is an abstract class which represents a Stateless Session Enterprise Java Bean. Each Stateless Session Bean must extend this class if it not extending something else. This avoids the need of each bean to implement methods from the **javax.ejb.SessionBean** class.
A home interface of an EJB defines the methods that allow a remote client to create, find, and remove EJB objects. It extends **javax.ejb.EJBHome**

An EJB's remote interface provides the remote client view of an EJB object. It defines the business methods callable by a remote client. The remote interface must extend the **javax.ejb.EJBObject** interface and corresponding business interface.

BusinessInterface is a marker interface. All business interfaces must extend this interface. Each EJB has a business interface which defines the enterprise Bean specific business methods. This is to put a compile time check on the methods exposed by EJB and methods implemented by EJB

3 Metadata Repository

3.1 Overview

One of the basic requirements of caB2B is to be able to download a UML model of any application from the caDSR and provide capabilities to build a query to fetch data from that data source. In order to understand the design of caB2B it is necessary to first understand the design and concept of the **m**eta**d**ata **r**epository (**MDR**).

MDR stores the metadata for an UML model including its semantic annotations like all CDEs including permissible values by decomposing the annotated UML model obtained from caDSR.

It also contains all-to-all paths between every two classes. The caB2B server pre-calculates the paths between all pairs of classes in the UML model and stores them in the MDR. Classes from different applications are connected based on their attribute's CDE match. This involves matching the concept codes of the classes and their attributes in order. Finally, given the amount of information it stores, it is also possible to get all the paths between two classes across two different UML models based on semantic interoperability.

The design of MDR is the basic foundation for caB2B backend. It enables the caB2B query engine to provide the following functionalities:

- Metadata search
- Auto generation of user interface for entering predicates
- Automatic path resolution between two query predicates
- Category support
- Inter model queries based on semantic joins

caB2B uses Dynamic Extensions framework to store the UML model along with its semantic annotations.

3.2 What is Dynamic Extensions?

Dynamic Extensions is a framework that allows creating business objects dynamically in the form of entities and attributes. Following are the **D**ynamic **E**xtensions (**DE**) terms regularly referred in this document:

- Entity is a UML class.
- Attribute is a UML attribute.
- Association is relationship between any two entities.

The metadata definition of entity and attribute includes:

- Model Properties (i.e. Data type, Precision etc.)
- Semantic properties (i.e. concept codes)
- Value domain specification (CDE public id, permissible values etc.)

For the detailed design of MDR, please refer to Section Overview and UML metadata of the Dynamic Extensions design document.

Note: Since Dynamic Extensions design document is not formally released, those two sections are appended to the Appendix of this document. Once the DE design document is release, the appendix will be deleted.

3.3 Storing UML model

This activity involves following

- Parsing the domain model downloaded from caDSR using caGrid APIs
- Storing the metadata in DE along with inheritance relations.
- Finding out semantic relations of entities from current model to entities already present in system (coming from different model)

The class diagram below shows all the classes involved in parsing domain models, storing them in MDR, and finding and storing all possible non-redundant paths.

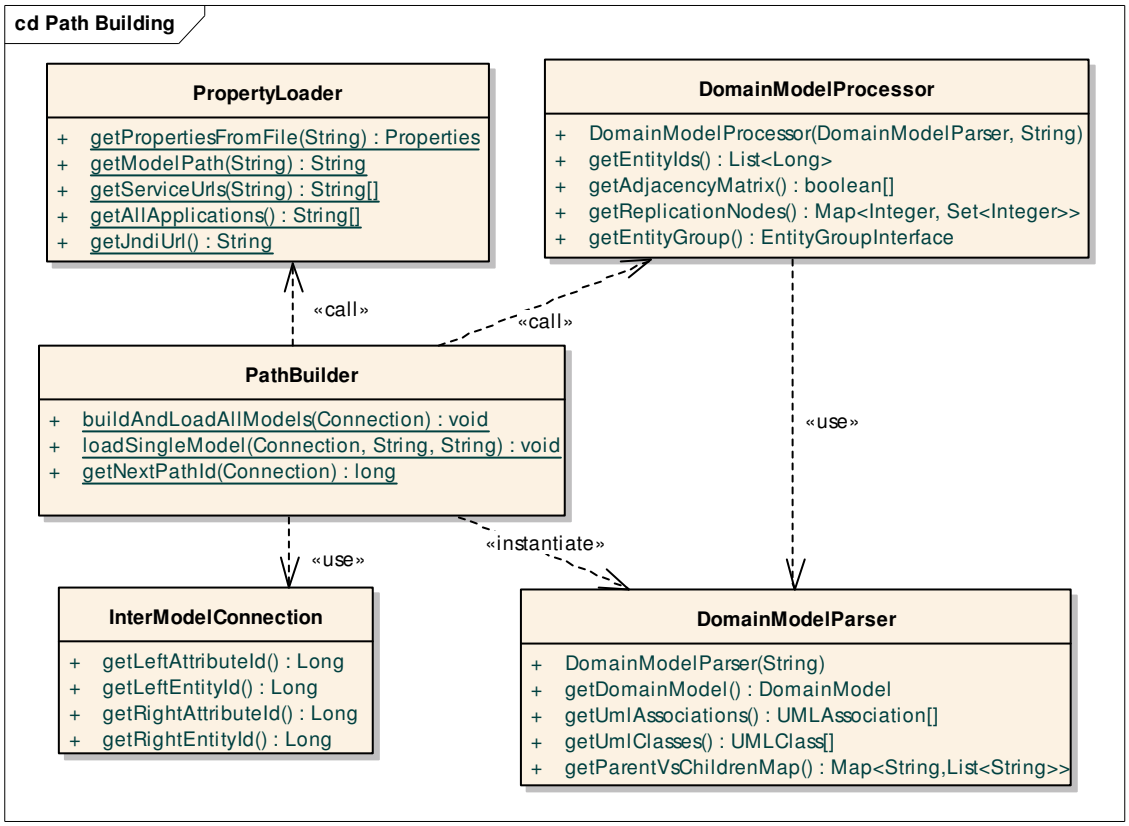

Figure 4 Classes involved in storing UML model to MDR

- **PathBuilder** is a controller that calls different utility classes to populate MDR by decomposing models defined in cab2b.properties file. It loads all possible non-redundant paths for a given model to database.
- **DomainModelProcessor** stores the decomposed UML model to MDR. It first transforms converts model into DE's objects and processes inheritance relationship in the model. Then DomainModelProcessor stores these objects to MDR. It also generates an adjacency matrix and related information required for path calculation. An instance of this class refers to one domain model
- **DomainModelParser** converts a domain model XML file located at a given path to caGrid metadata objects using the caGrid metadata utility (gov.nih.nci.cagrid.common.Utils).
- **PropertyLoader** handles fetching properties from "cab2b.properties" file. It provides methods
  - To get all the models loaded in caB2B

- To get the file system path for the domain model XML of a given application

- **InterModelConnection** represents one link present between two entities from different models. This link is a pair of semantically equivalent (i.e. reused CDEs) attributes of classes from different models.

3.4 Path Generation Module

3.4.1 Steps and Classes Involved

This module calculates all possible ways to connect any two entities in the same model. It consumes the adjacency matrix generated by **DomainModelProcessor.** It converts that to a **Graph** object which is an adjacency list representation of a (directed) graph. Each vertex of the graph is a **Node**. This module outputs set of *edu.wustl.cab2b.server.path.pathgen.Path* which is an immutable representation of a path as a collection of the following:

- Source/From edu.wustl.cab2b.server.path.pathgen.Node
- Destination/To edu.wustl.cab2b.server.path.pathgen.Node
- A java.util.List of intermediate nodes needed to traverse from `fromNode` to `toNode`.

At any point in time, **GraphPathFinderCache** contains all the paths between all pairs of nodes that have been computed till then. When the algorithm terminates, this cache will thus contain all the resultant paths. This cache helps avoid recalculation of paths between a pair of nodes, and thus improves efficiency. Figure below shows all the classes involved in this module.

PathReplicationUtil replicates paths of parent entity to its child. For example suppose **P1, C1, P2, C2** are classes. C1 is child of P1 and C2 is child of P2. There is a bi-directional association present between P1 and P2. There is not association between C1 and C2. Then system generates following paths along with normal path between P1 and P2

1. Path between P1 and C2
2. Path between C1 and P2
3. Path between C1 and C2

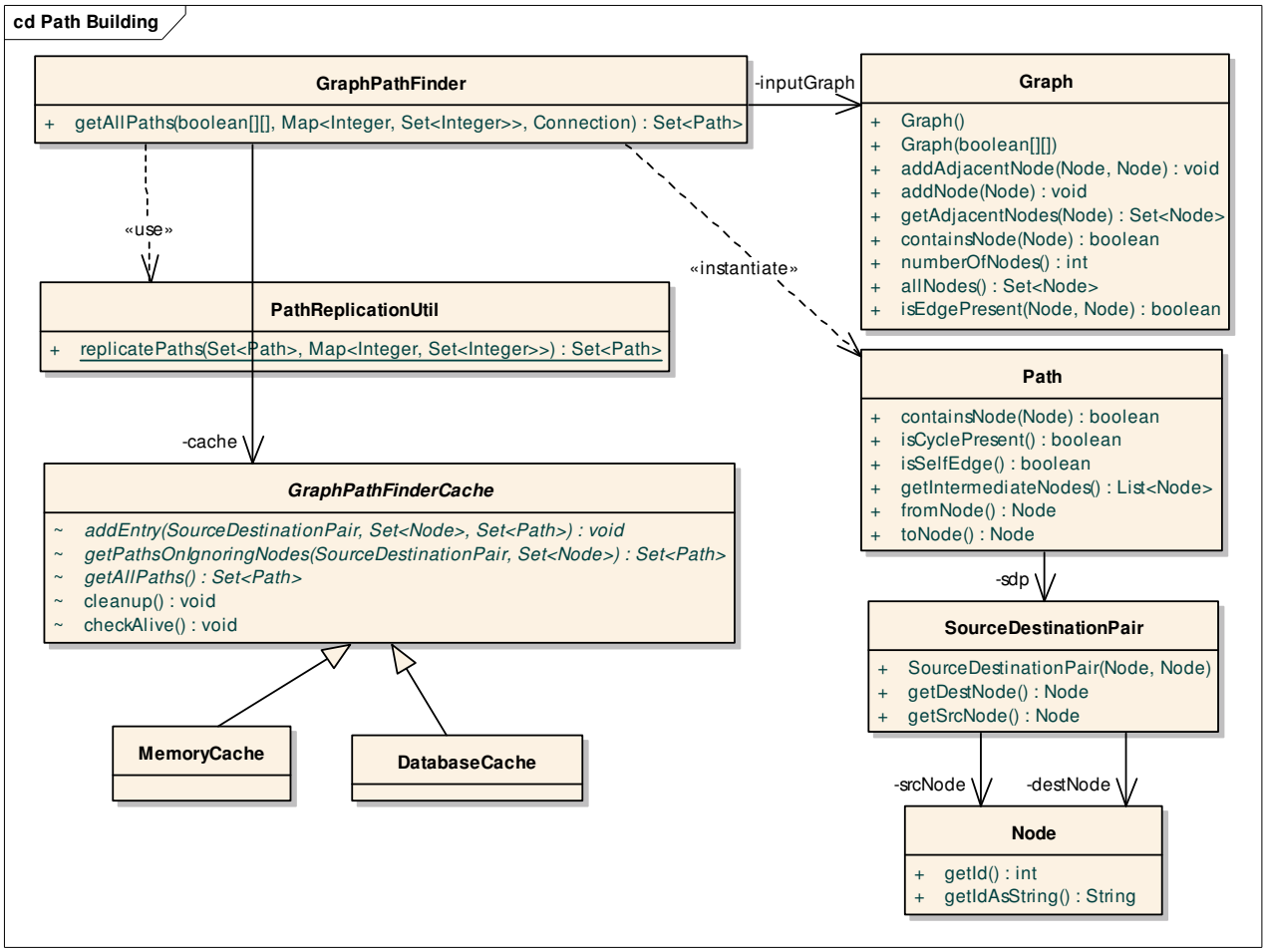

Figure 5 Class diagram of Path Generation Module

3.4.2 Algorithm for Path Generation

This algorithm computes all possible paths present in a directed graph. No path returned should contain a cycle. Suppose the graph is (V, E) where V is the set of vertices and E is the set of edges. A source-destination-pair (SDP) is represented as `i->j`.

`GraphPathFinderCache.getPathsOnIgnoringNodes(SDP, Set) method` returns the set of paths for given SDP and ignored nodes. Denote the SDP by `i->j`, and ignoredNodes by N.
Let `n(p)` denote the nodes in a path p. Then, given that `N1 ⊆ N2`, we can compute `P(i->j, N1` from `P(i->j, N2)` using the following formula
`P(i->j, N1) = {p : p ∈ P(i->j, N2), n(p) ∩ N1 = {} }.`
Thus this method is expected to do the following:

1. If there is an entry in the cache `P(i->j, N)`, return it, else continue.
2. If there exists an entry in the cache `P(i->j, M)` such that `M ⊆ N` then compute `P(i->j, N)` using above formula and return it, else continue
3. Return null

Note that if an empty set of paths is returned, it means that it has been computed already that there are no paths present, i.e. P(i->j, N) = {}. The algorithm is as follows:

For each pair of nodes `{i, j : i ∈ V, j ∈ V, i ≠ j}` in the graph, call `getPaths(i->j, {})`. Self-edges (a self-edge is a path of the form `i->i`) are then added to the resulting set of paths. `getPaths()` is the method where the core of the algorithm resides. Suppose `P(i->j, N)` is the set of paths about to be returned from `getPaths()`. Following is what happens on a call `getPaths(i->j, N)`, where N is the ignoredNodesSet :

1. Let `X = GraphPathFinderCache.getPathsOnIgnoringNodes(SDP, Set)` with `(i->j, N)` as parameters;

   If `X != null`, then `P(i->j, N) = X`; return `P(i->j, N)`.

   Else continue.

2. If `i->j ∈ E` then add a path `i->j` to `P(i->j, N)`.

3. Let `K = {k : k ∈ V, k ≠ i, k ≠ j, k ∈ N, i->k ∈ E)`.

   For each `k ∈ K`, do the following:

   1. Call `getPaths (k->j, N ∪ {i})`. Suppose the returned set of paths is R.
   2. For each path `Rx (0 < x < |R|) in R`, add the path `i->Rx to P(i->j, N)`.

4. Add `P(i->j, N)` to the cache.

5. Return `P(i->j, N)`

3.5 Category

3.5.1 What is a Category

Category is a collection of attributes from one or more UML classes. These UML classes may be from same or different applications. The UML classes in a category should be directly or indirectly connected using UML associations.
As an illustration of the usage of category, consider the following use case: Get all genes with annotation which are associated with a given "Gene" through pubMed literature abstract i.e. get list of genes having literature relationship correlation value > 0.5 and have relationship with given gene. The UML diagram for the classes in the query is

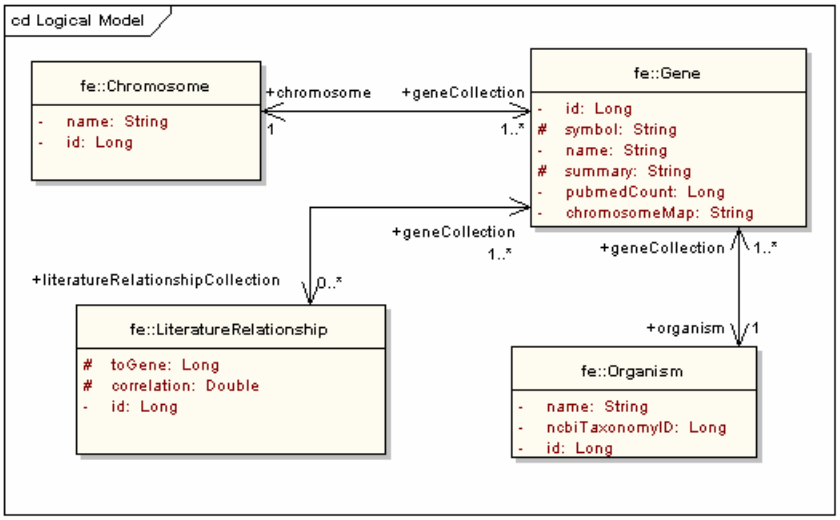

Figure 6 diagram for classes in category

To build the example query, user would

1. Search the four classes individually
2. Add limits on each of them
3. Connect all the classes in the DAG view

Shortcomings of above process:

- UML Class is a collection of attributes that makes sense to developers and bioinformaticians.
- The steps described above are cumbersome and time-consuming
- Each user who wishes to perform this query has to follow this process every time

In certain cases it may be found/felt that each user will define limits on specific attributes of certain logically related classes and connect them by similar paths. In such cases, those attributes can be grouped together to build predefined units with unambiguous paths to save users' time. These predefined units are categories.

Benefits

- Ability to apply limits on attributes of several UML classes in one go
- Paths among classes in a category will be predefined in metadata. Thus, the user need not find paths required to traverse logically related classes every time.
- End-user sees attributes in a single logical unit even though they belong to different classes due to modeling constraints
- Users with limited knowledge of UML domain models can query on categories.
- Advanced users can also use categories as building blocks for their complex queries

3.5.2 Creating a Category

Category is defined as a well-formed XML file called category XML. **CategoryXmlParser** parses this file and generates an **InputCategory** object. **PersistCategory** converts InputCategory to **Category** hibernate-object which will be saved by **CategoryOperations.** This flow in explained in sequence diagram below

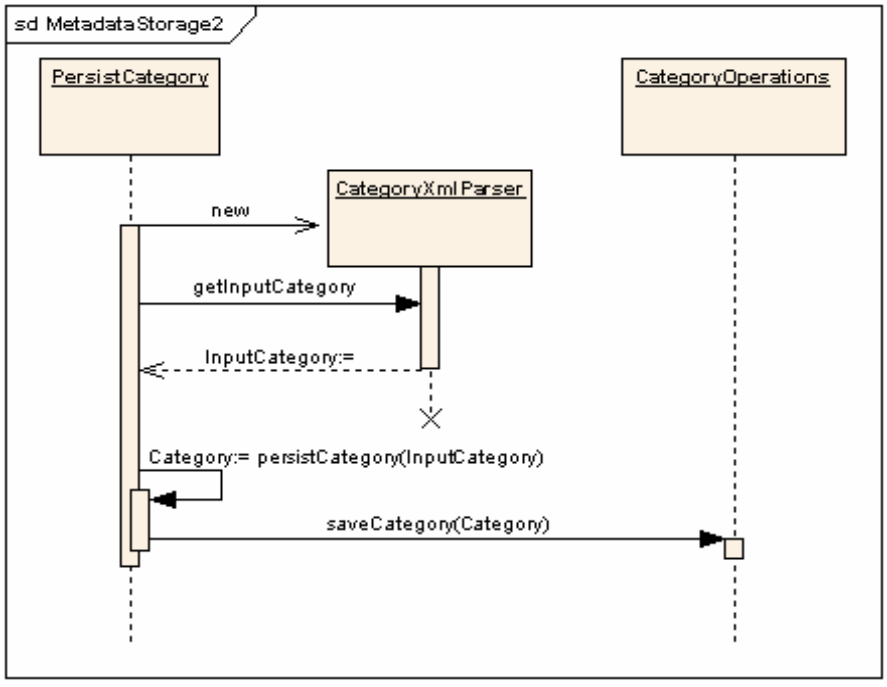

Figure 7 Sequence diagram saving a category

Category XML

This is a well-formed XML file, which defines a category. All categories are first defined as a Category XML and then they are imported into the caB2B MDR. The structure of this file is as follows:

```
<?xml version="1.0" encoding="windows-1250"?>
<Category>
    <CategorialClass name="" IdOfPathFromParentToThis="-1"> <!-- root -->
        <Attribute name = />
        <Attribute name = />
        .
        .
        .
        <CategorialClass name="" IdOfPathFromParentToThis = "23451">
          <Attribute name = />
          <Attribute name = />
          .
          .
          <CategorialClass name="" IdOfPathFromParentToThis = "53451">
            .
                .
                    .
          </CategorialClass>
        </CategorialClass>
        .
        .
    </CategorialClass>
    <!--  here are SubCategories  -->
    <Category>
    .
    .
    </Category>
</Category>
```

Figure 8 Category XML structure

Below is the example of the Category XML file for the category “Genomic identifiers”

```
<?xml version="1.0" encoding="windows-1250"?>
<Category name="Genomic Identifiers">
  <CategorialClass name="edu.wustl.geneconnect.domain.Gene" IdOfPathFromParentToThis="-1"> <!-- root -->
    <Attribute name = "ensemblGeneId" displayName = "ensemblgeneID"/>
    <Attribute name = "unigeneClusterId" displayName = "uniGeneClusterId"/>
    <Attribute name = "entrezGeneId" displayName = "entrezGeneId"/>
    <CategorialClass name="edu.wustl.geneconnect.domain.MessengerRNA" IdOfPathFromParentToThis = "716">
      <Attribute name = "ensemblTranscriptId" displayName = "mRNAEnsemblTranscriptId"/>
      <Attribute name = "genbankAccession" displayName = "mRNAgenBankAccessionNumber"/>
      <Attribute name = "refseqId" displayName = "mRNArefSeqId"/>
      <CategorialClass name="edu.wustl.geneconnect.domain.Protein" IdOfPathFromParentToThis = "616">
            <!-- Path -> Gene-mRNA-protein -->
            <Attribute name = "ensemblPeptideId" displayName = "ensemblPeptideId"/>
            <Attribute name = "refseqId" displayName = "proteinRefSeqId"/>
            <Attribute name = "uniprotkbPrimaryAccession" displayName = "proteinUniProtKBPrimaryAccession"/>
            <Attribute name = "genbankAccession" displayName = "proteinGenBankAccession"/>
      </CategorialClass>
    </CategorialClass>
  </CategorialClass>
</Category>
```

Figure 9 Example of Category XML file

3.5.3 Class Diagram

Classes involved in category creation are shown in figure shown below.

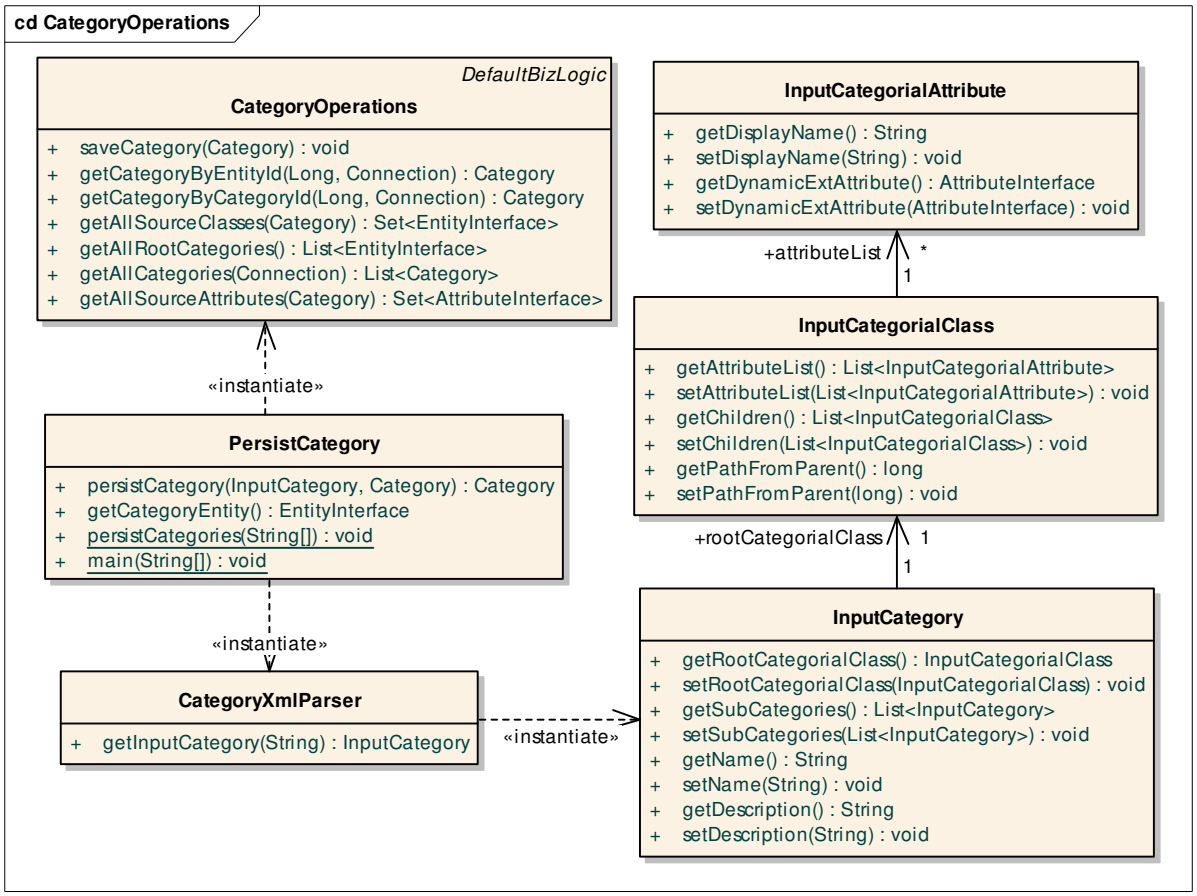

Figure 10 Classes involved in category creation

- **InputCategory** is an object representation of the "**Category**" tag of category XML.
- **InputCategorialClass** is an object representation of the "**CategorialClass**" tag of category XML.
- **InputCategorialAttribute** is an object representation of the "**Attribute**" tag of category XML.
- **CategoryXmlParser** provides methods to parse a category XML (see references) file and converts it into Java object form. These Java objects will be used in actual category creation and saving.
- **PersistCategory** provides methods to save a category in the database. It uses CategoryXmlParser to convert a category XML to corresponding objects (**InputCategory**) and then builds actual Category objects and saves them to the database using Hibernate.
- **CategoryOperations** provides functions for database operations needed for a category such as save and retrieve.

3.6 Metadata Cache

Contents of MDR are needed frequently by various cab2b-components. To improve efficiency, by avoiding database calls, metadata cache module is introduced. Classes involved in this module are shown in diagram below.

IEntityCache is an interface with methods needed for metadata search. Those will be explained in Metadata Search section later. **AbstractEntityCache** is an abstract class having all the methods exposed by this module. All components access MRD information

through this class only. It provides variety of methods to get metadata along with providing searching methods from IEntityCache. g*etCab2bEntityGroups ()* is the only abstract method in AbstractEntityCache. This method is used to populate the cache. So it is up to implementer's responsibility to decide how it will get entity groups. There are two implementing classes **EntityCache** and **ClientSideCache**.

EntityCache calls dynamic extension API directly to get entity groups. EntityCache is a singleton class residing in server side. It is instantiated and populated on first server call. It is then used by all of the components running at server side. **ClientSideCache** calls an EJB UtilityBean to get entity groups as it won't have direct access to DE APIs. It is also a singleton class which is instantiated and populated before launching client. It is then used by all of the components running at client side.

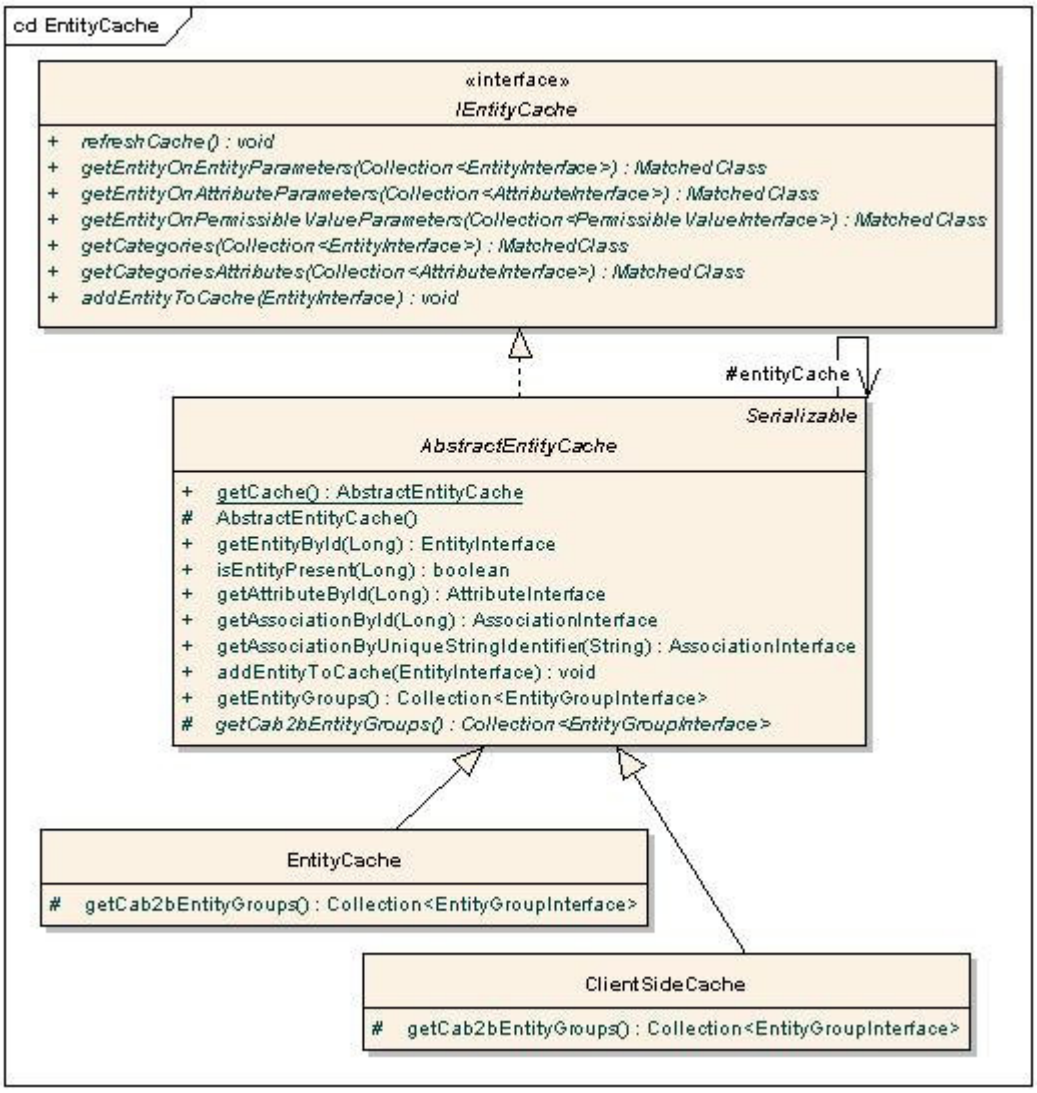

Figure 11 Classes in Metadata cache module

4 Metadata Search

4.1 Overview

As the end users may not be familiar with object models, there should be a way for them to first search for the entity on which they want to query. For example, an end user will not know which entity has the attribute for clinical diagnosis in the caTissue object model. The metadata allows users to first search for entities based on metadata such as names, attribute names, permissible values, or definitions using free text search or concept codes. This module has backend search implementation and a user interface to specify search conditions, display search results.

4.2 Backend Implementation

Metadata search back end part exposes one method on **MetadataSearch** class
search(int[] searchTarget, String[] searchString, int basedOn) Parameter details are:

- basedOn: the basis of search, whether a text based or concept code based search is asked
- searchString[]: Array of Strings created by splitting string entered by user based on space characters
- searchTraget[]: Where to search is specified by this. Typical values are class, attribute, permissible values, class-description and attribute-description

The dataset to be searched is decided by the **IEntityCache** object passed to construct MetadataSearch object. IEntityCache provides searching methods like

- getEntityOnEntityParameters(entityCollection)
- getEntityOnAttributeParameters(attributeCollection)
- getEntityOnPermissibleValueParameters(PVCollection)
- getCategories(Collection<EntityInterface> entityCollection)
- getCategoriesAttributes(attributeCollection)

Each of above method returns a **MatchedClass** object. MatchedClass is a wrapper around set of entities. The **search ()** method searches each searchTarget for all strings in searchString array by calling one of the above methods of IEntityCache for each searchString. Then it merges the results of all individual searches using method *createResultClass ()* and returns one **MatchedClass** object.

CompareUtil is responsible for deciding whether a particular entity, attribute, permissible value or semantic property is matching user criterion. It has **compare()** methods which take pair of entities, attributes, permissible values or semantic properties and returns a boolean. If user entered string is contained in string to be search, then it is added to result. Below diagram shows all of the classes along with their behaviors and relationships with each others.

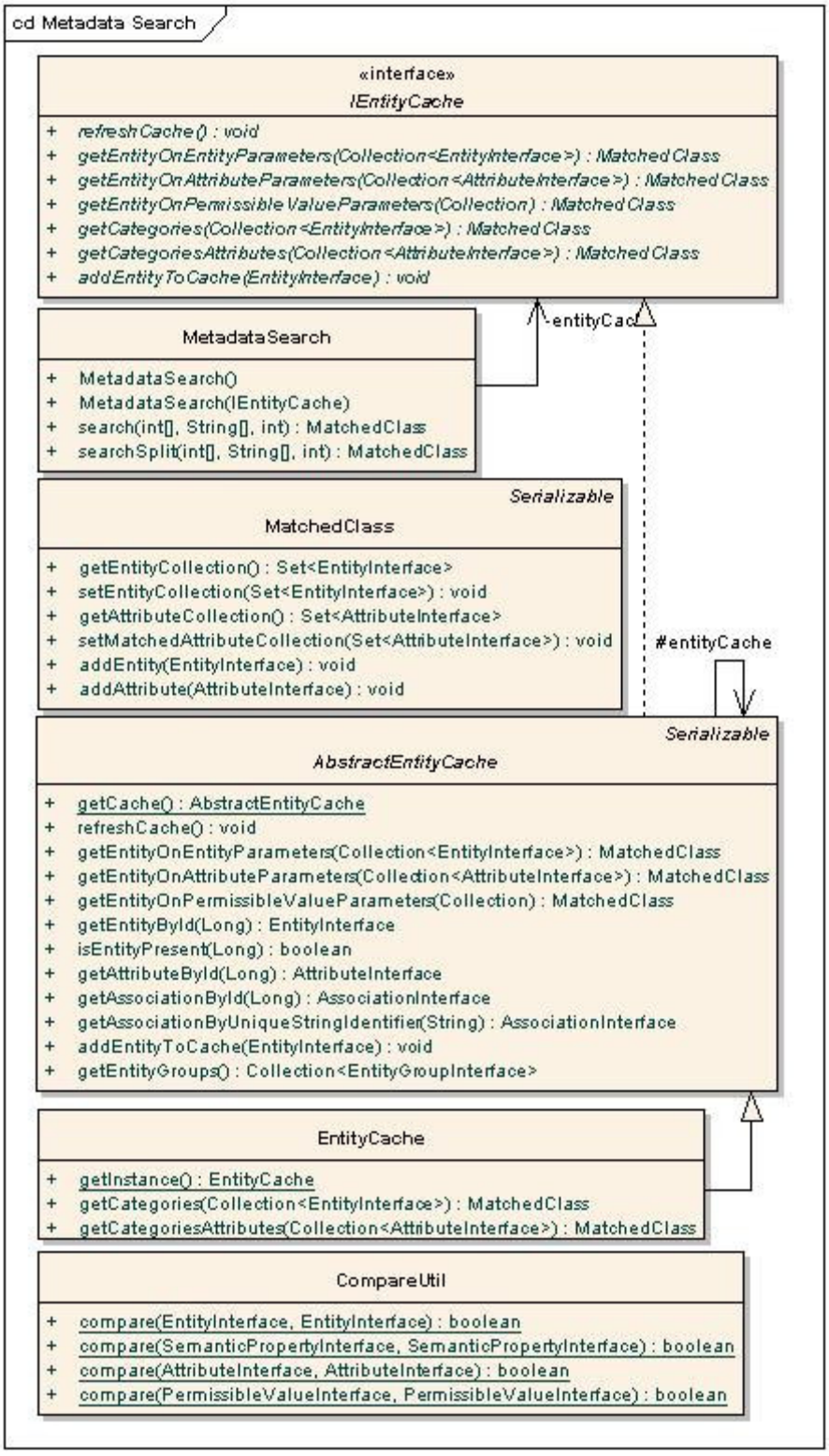

Figure 12 Classes- Metadata Search backend

4.3 User Interface

User interface of this module mainly consist of **SearchPanel**. AdvancedSerachPanel and SearchResultPanel are embedded in SearchPanel for common functionalities and code reuse. **AdvancedSearchPanel** is panel where user specifies search criterion, **SearchResultPanel** displays search results using pagination component (for details refer to chapter **Pagination Component**). The diagram below shows these classes along with their local classes.

- **AdvancedSearchPanel:** It is a class which contains commonalities between the collapsible portions of the advanced/category search panels for the 'Choose Category' as well as 'AddLimit' section from the main search dialog. TaskPaneMouseLister is its local class. The collapsible portion provides options for searching category, attribute, permissible values with provision of concept code or text search.
- **SearchResultPanel:** This class that contains commonalities required for displaying results from the 'AddLimit' and 'Choose Category' section from the main search dialog. MyCellRenderer, AddLimitButtonListner, CDETableModel, EditLimitButtonListner, AttributeDetailsLinkListener are its local class.
- **SearchPanel:** It is the main class that contains UI commonalities between the advanced/category search panels for the 'Choose Category' as well as 'AddLimit' section from the main search dialog. SearchActionListener is its local class.

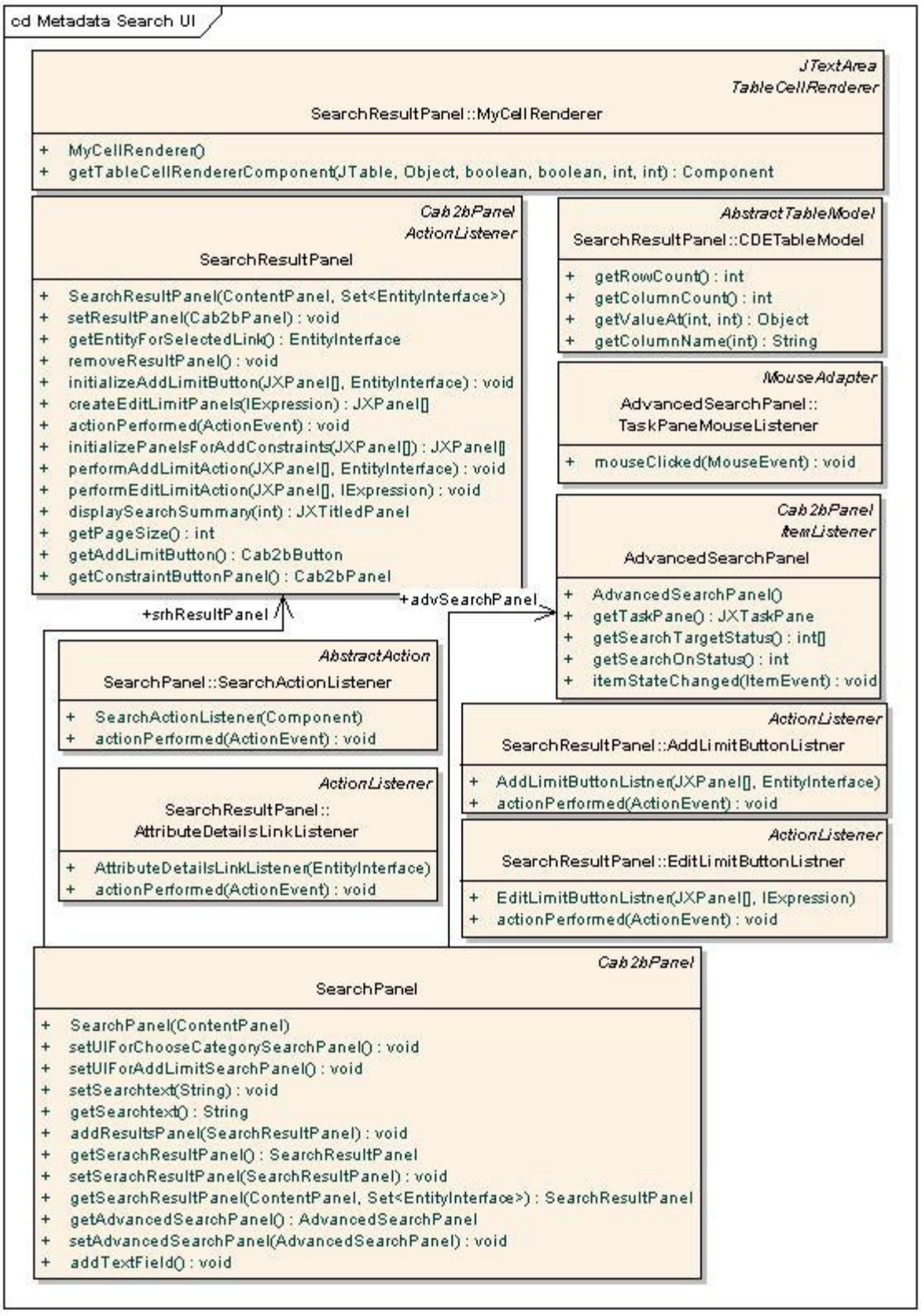

Figure 13 Classes- Metadata Search user interface

5 Query Object

5.1 Overview

The **query-object** (*IQuery*) provides the interfaces used to represent a user-defined query. The query consists of **outputs** (represented by *IOutputTreeNode*) and **constraints** (represented by *IConstraints*). User defined **conditions** (e.g. Participant.sex = 'female') are represented by *ICondition.* Conditions on different attributes of an entity are grouped together as a **rule** (represented by *IRule*). Various rules/expressions on an entity can be logically grouped into an **expression** (represented by *IExpression*). An expression thus consists of operands (i.e. rules or sub expressions; this is represented by *IExpressionOperand*) connected by logical operators (AND, OR). Operands in an expression may also be parenthesized.

The various expressions thus formed need to be linked together. Two expressions are linked by an **association** (represented by *IAssociation)*. These linkages among the expressions constitute the **join graph** (represented by *IJoinGraph*).

Interface Summary	
ICondition	A condition containing an attribute, relational operator and value(s). E.g. participant.sex = 'Male' forms one ICondition
IRule	A list of conditions on different attributes of an entity. The conditions in a rule are implicitly linked by an AND condition.
ILogicalConnector	Represents a logical connector (AND / OR). The nesting represents the number of parentheses (depth of parentheses) around the logic portion (AND or OR) of the connector.
IExpressionId	An immutable wrapper around int used to uniquely identify an expression within a query. It is auto generated when an expression is added to a query (using IConstraints.addExpression).
IExpression	A list of operands, and the logical connectors (AND, OR) together form a logical expression. The connectors are identified by the position of the operands on either side. An IExpression belongs to a constraint entity and constraints on another associated entity will be present as a sub expression on the associated entity. Conversely, if an expression has a sub expression, there must an association in the join graph from the parent expression to the sub expression. ***Note***: "sub expression" refers to an operand that is the IExpressionId of the child expression. The entity of the sub expression will generally be different from the entity of this expression (the exception is when a class is associated to itself, e.g. Specimen class in caTissue Core). The expression for an ExpressionId is found from IConstraints.
IExpressionOperand	A marker interface for an operand. An operand is either a sub expression (in which case, the corresponding expression id is added), or a rule.
IJoinGraph	A rooted, directed acyclic graph with expressions as vertices, and associations as edges. The graph will always contain all the expressions' ids (obtained from IConstraints) as vertices. The vertices will be added to/removed from the Joingraph as and when expressions are added to/removed from IConstraints. The methods in Joingraph can only add/remove associations among the vertices. If v1 and v2 are two vertices, the direction will be v1->v2 if v2 is a sub expression of v1. This graph determines the join conditions in the query. E.g. for each edge (v1, v2) there will be a join between the entities (IConstraintEntity) of the expressions denoted by v1 and v2;

	and the join condition is determined by the information in IAssociation.
IQueryEntity	An entity on which the user specifies limits (constraints) e.g. Participant is an IQueryEntity
IConstraints	Contains information about the constraints of a query. It contains a list of IExpressions. This list is indexed by IExpressionId. This is global storage for all the expressions in a query. Calling the addExpression() method here creates an IExpression. It also contains a join graph for specifying how the expressions are linked together.
IOutputEntity	An entity which is desired as the output of the query.
IOutputTreeNode	The output entities of a query form a tree with vertex as IOutputEntity and edge as IAssociation. IOutputTreeNode represents one node of this tree.
IQuery	The query object representing a complete user-defined query consisting of outputs and constraints.

5.2 Class diagram

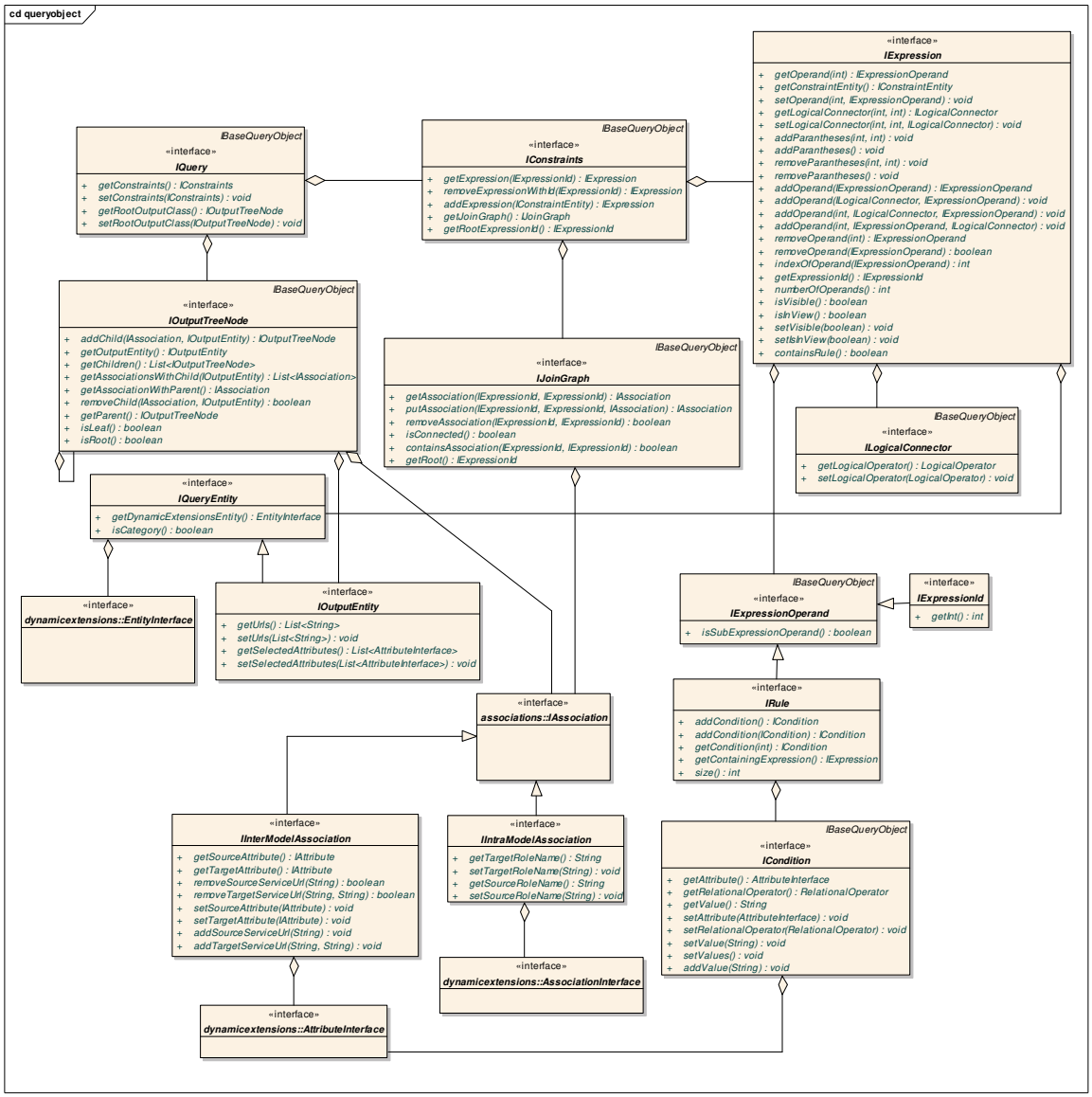

Figure 14 Interfaces that compose the query object

6 Query Engine

Query engine interprets the query object and converts it to DCQL(s), executes DCQL(s) and gets result back from data services

6.1 Overview

The category constraints made by the user using the caB2B client's DAG view are stored in the query-object i.e. *ICab2bQuery* (which extends *IQuery* to add information regarding output class' service URLs). The query engine is an EJB that processes the *ICab2bQuery* to form the corresponding DCQL, executes the DCQL, and returns the results back to the client.

6.2 Class diagram

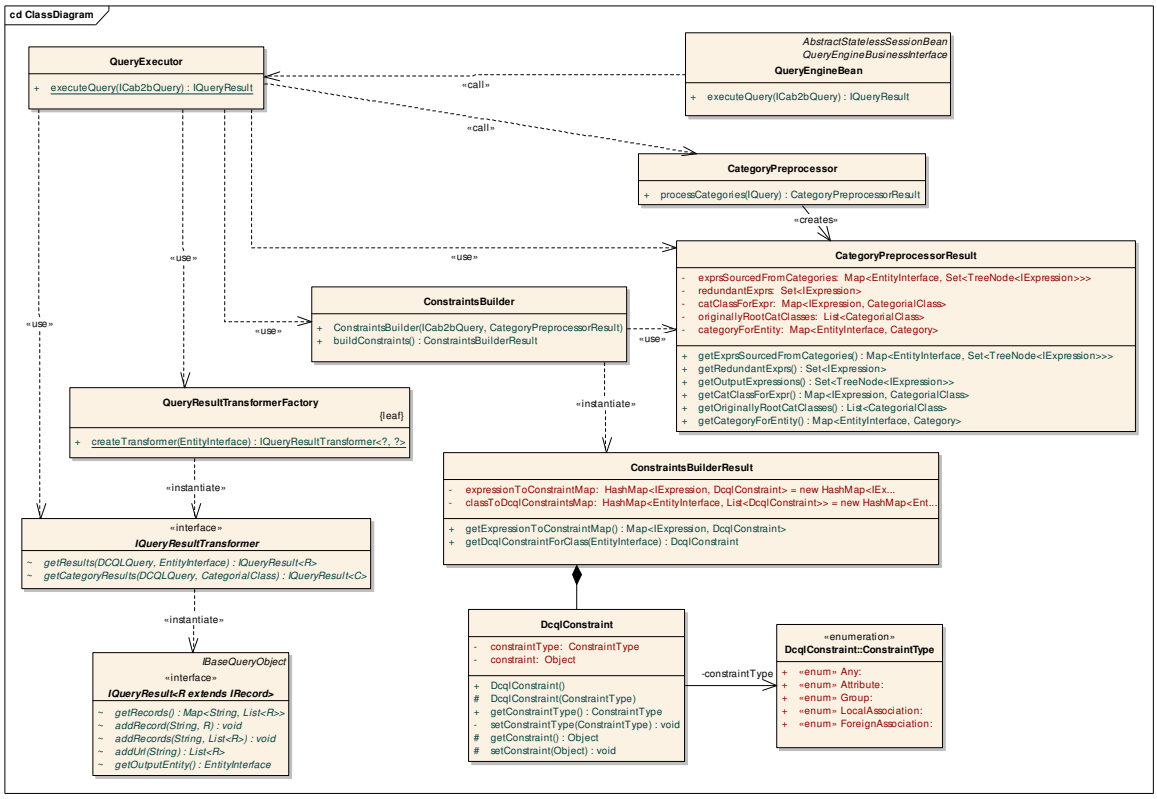

Figure 15 Interfaces and classes that compose the query engine

Description of classes and their interactions:

- ***QueryEngineBean*** is an EJB that receives the calls for query execution from the caB2B client. It just forwards the call to QueryExecutor.
- ***IQueryResult*** is a map of the service URL to records obtained from that service. The records are represented as a two-dimensional array with columns corresponding to attributes and rows to values.
- ***QueryExecutor*** uses the *ConstraintsBuilder* to form DCQL(s), hands over the DCQL(s) to an appropriate transformer and returns the resulting *IQueryResult*. Multiple DCQLs are fired when the output is category; *CategoryPreprocessorResult* is used in this process.
- ***CategoryPreprocessor*** modifies an input *IConstraints* by expanding the *IExpression*s on categories to its constituent classes. Thus *IConstraints* is modified to only contain *IExpression*s on classes.

- ***CategoryPreprocessorResult*** represents the results of the ***CategoryPreprocessor.*** It provides additional information about the relationship between the original category entities in the query and the new expressions created for them.
- ***ConstraintsBuilder*** processes an *IQuery* object and returns a corresponding *ConstraintsBuilderResult* object (See Figure 13.0). It uses the ***CategoryPreprocessorResult*** for this processing.
- ***DcqlConstraint*** is a wrapper around any of the following four types of objects that compose part of the caGrid *DCQLQuery*. For details related to these and *DCQLQuery* please see the caGrid Programmer's Guide
  - *Attribute*
  - *Association*
  - *ForeignAssociation*
  - *Group*
- ***ConstraintType*** is used to distinguish among the above four types of constraints.
- ***ConstraintsBuilderResult*** provides the *DCQLConstraint* corresponding to each *IExpression* in the query.
- ***QueryResultTransformerFactory*** *provides* the appropriate transformer.
- ***IQueryResultTransformer*** executes the DCQL using the caGrid FQP and transforms the results to appropriate *IRecord*. See Query Result Transformers (Chapter ***Record Customization***).

6.3 Sequence diagram

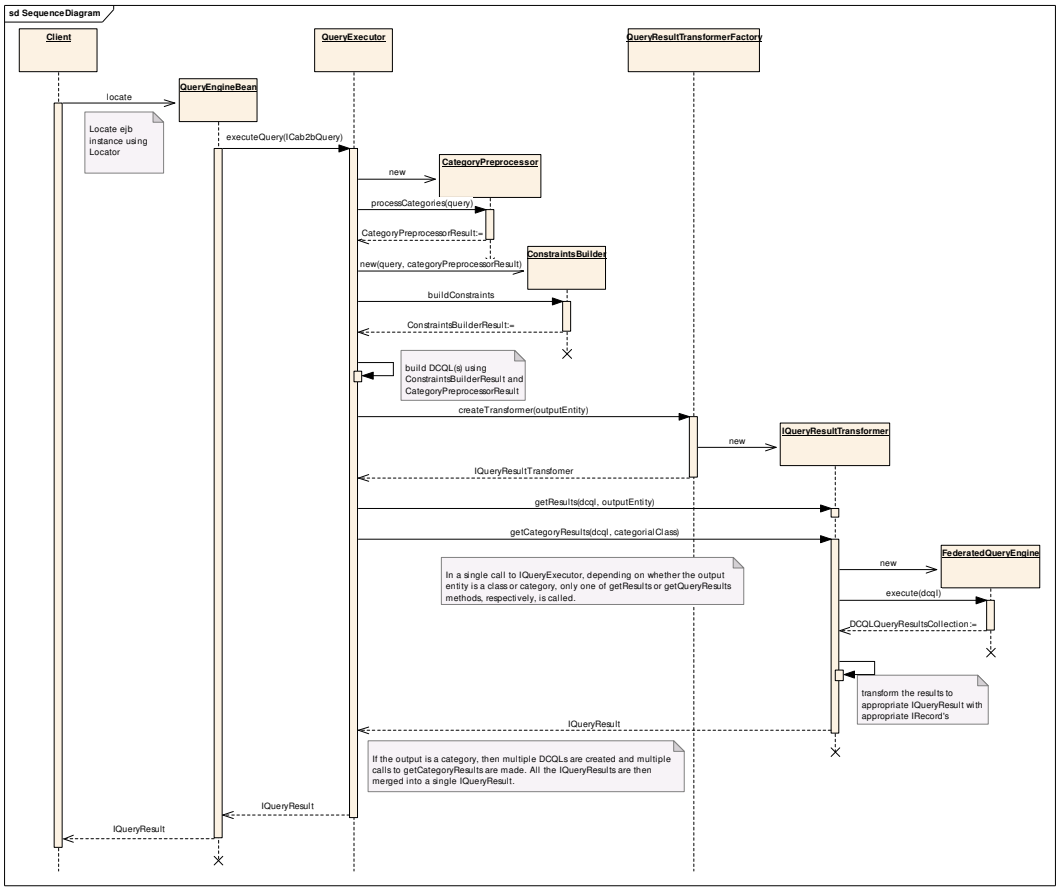

Figure 16 Sequence diagram to show how a query is executed and results are returned

6.4 Flowchart

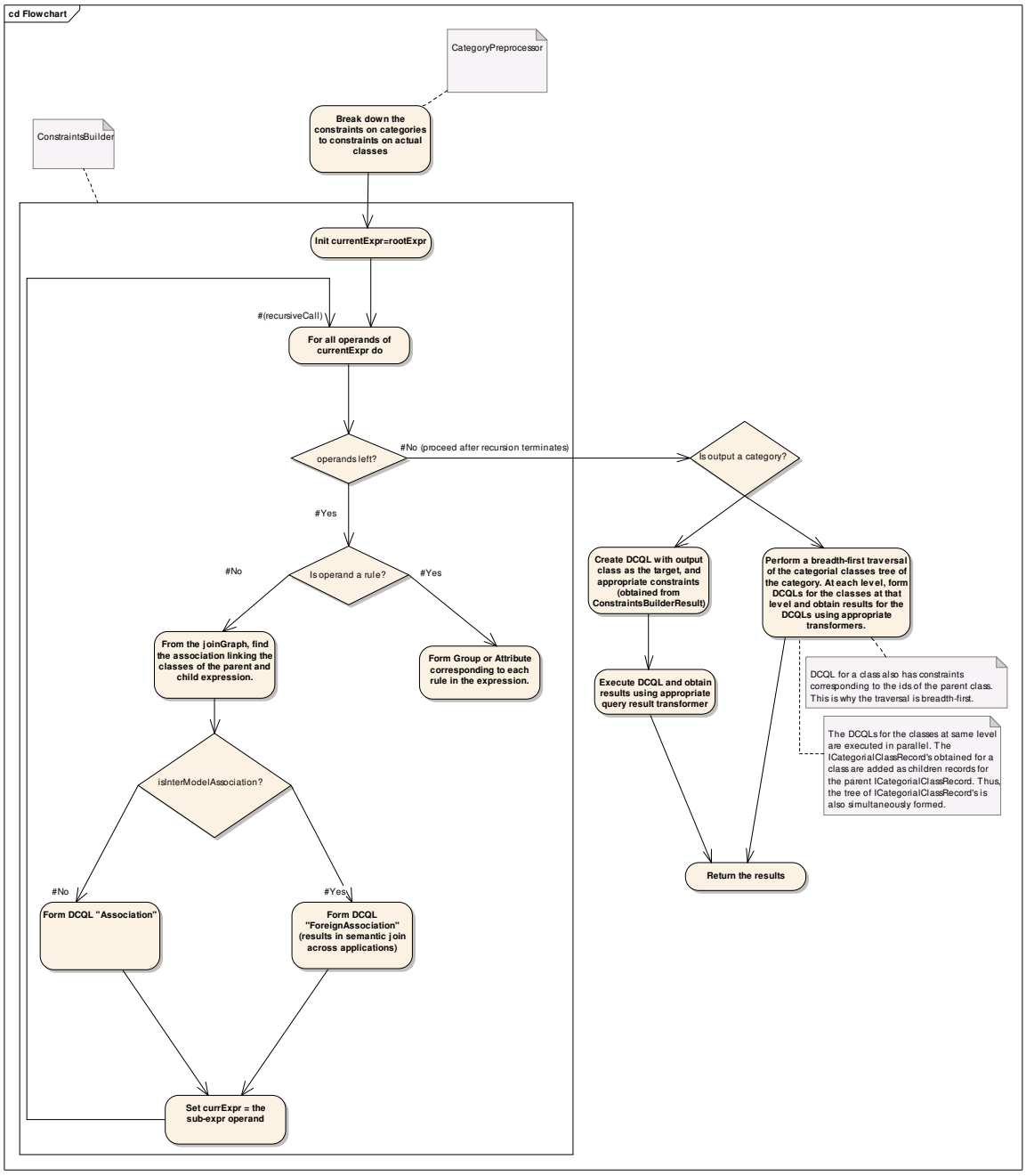

Figure 17 Detailed steps within the QueryExecutor

6.5 Lazy initialization

Sometimes a record may be very big i.e. it may consume a lot memory. An example is a biodatacube. Sending the complete record from the caB2B server to the client would be unreasonable in such cases because:

- Client-side memory would be relatively lesser.
- The user may not wish to see the complete record; only some parts of it may be required at a time.

- Client performance would deteriorate due to the large amount of network traffic.

Thus, it is required, in some cases, to be able to initialize a record lazily.
Lazy initialization entails the following:

1. Store the complete record on the server side, and provide a handle to it. This is done by *edu.wustl.cab2b.server.queryengine.LazyInitializer.* A complete record is represented by the interface *edu.wustl.cab2b.common.queryengine.resul.IFullyInitializedRecord.*
2. Maintain the handle as part of a partially initialized record. A partially initialized record is represented by the interface *edu.wustl.cab2b.common.queryengine.result.IPartiallyInitializedRecord.*
3. Obtain data for the uninitialized portions by providing a handle to the fully initialized record, and parameters that identify the portions to fetch. The **lazy parameters** are represented by *edu.wustl.cab2b.common.queryengine.result.ILazyParams;* the method that does this lazy initialization is *LazyInitializer.getView().*

Details of these interfaces and classes follow:

edu.wustl.cab2b.server.queryengine.LazyInitializer

- ***int*** *register(IFullyInitializedRecord fir)* Registers a fully initialized record, and provides a handle to it. Currently, the record is stored in an in-memory map.
- *IPartiallyInitializedRecord getView(**int** handle, ILazyParams params)* Identifies the fully initialized record corresponding to the handle, and requests it to provide the *IPartiallyInitializedRecord* corresponding to given *ILazyParams.*
- *IFullyInitializedRecord getFullyInitializedRecord(**int** handle)* Returns the *IFullyInitializedRecord* corresponding to the handle.
- ***void*** *unregister(**int** handle* Unregisters the record. Currently, the record is removed from the in-memory map.

edu.wustl.cab2b.common.queryengine.result.IFullyInitializedRecord:

- *IPartiallyInitializedRecord view(ILazyParams params, **int** handle)*
  Returns the partially initialized record that corresponds to the parameters. The handle is passed on to the newly created *IPartiallyInitializedRecord.*

edu.wustl.cab2b.common.queryengine.result.IPartiallyInitializedRecord

- ***int*** *handle()* The handle to the fully initialized record.
- *ILazyParams initializationParams()* The parameters with which this record was created.

Example – *BioAssayData*
A *BioAssayData* record contains a biodatacube which is a three-dimensional array. The dimensions of this array can be quite large, and thus the cube can require a huge amount of memory. Lazy initialization of this record is done by lazily initializing the contents of this array based on the indexes of the cells the user is viewing.

Following interfaces thus represent the partially and fully initialized records respectively:

- *cab2b.common.caarray.IPartiallyInitializedBioAssayDataRecord*
- *cab2b.common.caarray.IFullyInitializedBioAssayDataRecord*

Refer to Record Customization for the class diagram showing the genealogy of these interfaces. The lazy parameters for this scenario are represented by **LazyParams** from *package edu.wustl.cab2b.common.queryengine.result.I3DDataRecord*

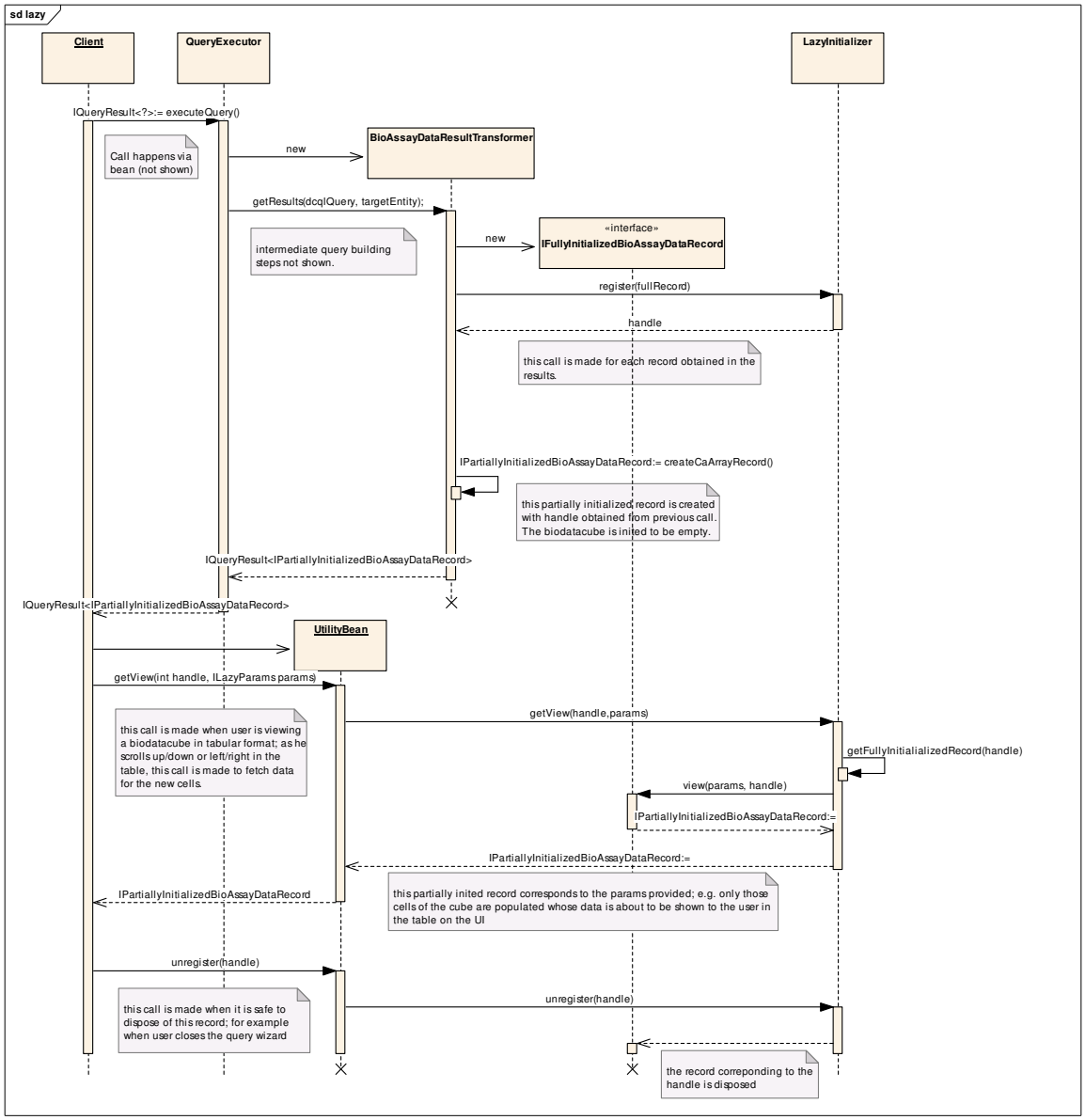

Figure 18 Sequence diagram - Lazy Initialization

7 Custom UI Components

7.1 Overview

The usual practice of UI development is to use standard UI controls with their default properties and behavior. The problems with this approach are:

1. If the application requires a property (e.g. font for labels) to be standardized across the application a change is needed at every place wherever that component instantiated. This is quite cumbersome.
2. Sometimes standard component doesn't provide required functionality or provides limited functionality

These are avoided by creating several components (see table) by customizing and extending the standard Swing and SwingX components. Customization includes modifying some default property and/or behavior for the standard component to suit the requirements.

The Usability Engineering group makes UI standards available. For example, all button labels should be of 'Arial 10pt Normal'. This is achieved by defining 'Cab2bButton' that extends 'JButton' and setting the font at the time of creation. So whenever a 'Cab2bButton' is created, it comes with customized font by default. Also, font can be changed very easily by only modifying 'Cab2bButton' class and the change would be reflected across all buttons in the application.

7.2 List of customized components

Original Component	**Customized Component Name**	**Customization details**	**Usability related?**
JButton	Cab2bButton	Default font and preferred size is set	Yes
JLabel	Cab2bLabel	Default font, background color is set	Yes
JCheckBox	Cab2bCheckBox	Default font, background color is set	Yes
JComboBox	Cab2bComboBox	Default font, background color and preferred size are set.	Yes
JRadioButton	Cab2bRadioButton	Default font, background color is set.	Yes
JTextField	Cab2bTextField	Default preferred size set.	Yes
JFormattedTextField	Cab2bFormatted-TextField	Customized to handle field validation like, field accepts only positive integer, floats and alphanumeric strings.	No
JXHyperlink	Cab2bHyperlink Cab2bHyperlinkUI	Customized by default to show the hyperlink text underlined and each hyperlink associated with a user object. Default visited and un-visited hyperlink color is set as per recommendation.	Associating user object with hyperlink is application specific.
JXDatePicker	Cab2bDatePicker	Default preferred size is set.	Yes
JXPanel	Cab2bPanel	Panel background color is by default se t to white. It can be changed to any other color by passing the appropriate Color object.	Yes
JXTitledPanel	Cab2bTitledPanel	This panel background color is by default set to white.	Yes

TableModel	LazyTableModel	Added the ability to fetch table data as and when needed to display huge data in table form.	No
JXTable	Cab2bTable Cab2bDefaultTable Model	By default, “select all” of table rows are enabled. Shows long texts in a text area with text wrapping.	No
	StackedBox	Customized to look as per the Visual Design specification.	Yes

Table 1 customized components in the caB2B application

7.3 Lazy Table Model

This is component developed for visualization of huge spreadsheet data. It only fetches data that is currently required. It additionally caches the data to improve performance. Classes involved are

- **LazyDataSourceInterface**: This is used by **LazyTableModelInterface** to fetch the data only when required. It provides the description of the data like number of rows, number of columns etc. The method getData(int row,int column) has a responsibility to fetch the data from the source (that may be a servlet, ejb or anything). Additionally it can cache the data.
- **AbstractLazyDataSource:** provides a sample implementation of the getData method. It converts the location of the required data to the cacheable page unit. Then fetches the data of the page from the data source, caches the page and extract required data from the page.
- **BDQDataSource:** This is the implementation of the AbstractLazyDataSource for bio data cube (BDQ) object. It provides implementation to fetch the portion of data for BDQ from the server and to extract the required data from the page. It converts x and y co-ordinate into the dimensions of the 3D representation of the BDQ object.
- **PageDimension** is used by data source to paginate the data. It gives the dimension of the pages of a particular data. **Page** represents a small block of data. The huge data can be broken down into the pages. **PageInfo** represents metadata about the page. It provides information like starting x and Y coordinate of a page in original data.
- **LazyTableModelInterface**: This is marker interface for the table models that uses LazyDataSourceInterface to fetch actual data.
- **DefaultLazyTableModel:** delegates all calls to the contained data source.
- **CacheInterface**: provides method to cache the pages of the data. This is used by data source.
- **BucketCache**: This is implementation of the cache based on bucket of the pages. It keeps the fixed bucket of the pages. Each page can go into a particular bucket depending on its coordinate. If a particular bucket is already occupied, the new page replaces existing page from that location.
- **MatrixCache**: This is similar to bucketCache with two dimensional bucket structures. Pages are put into a particular location of the matrix based on its coordinate. If it is already occupied the new page replaces existing page from that location.

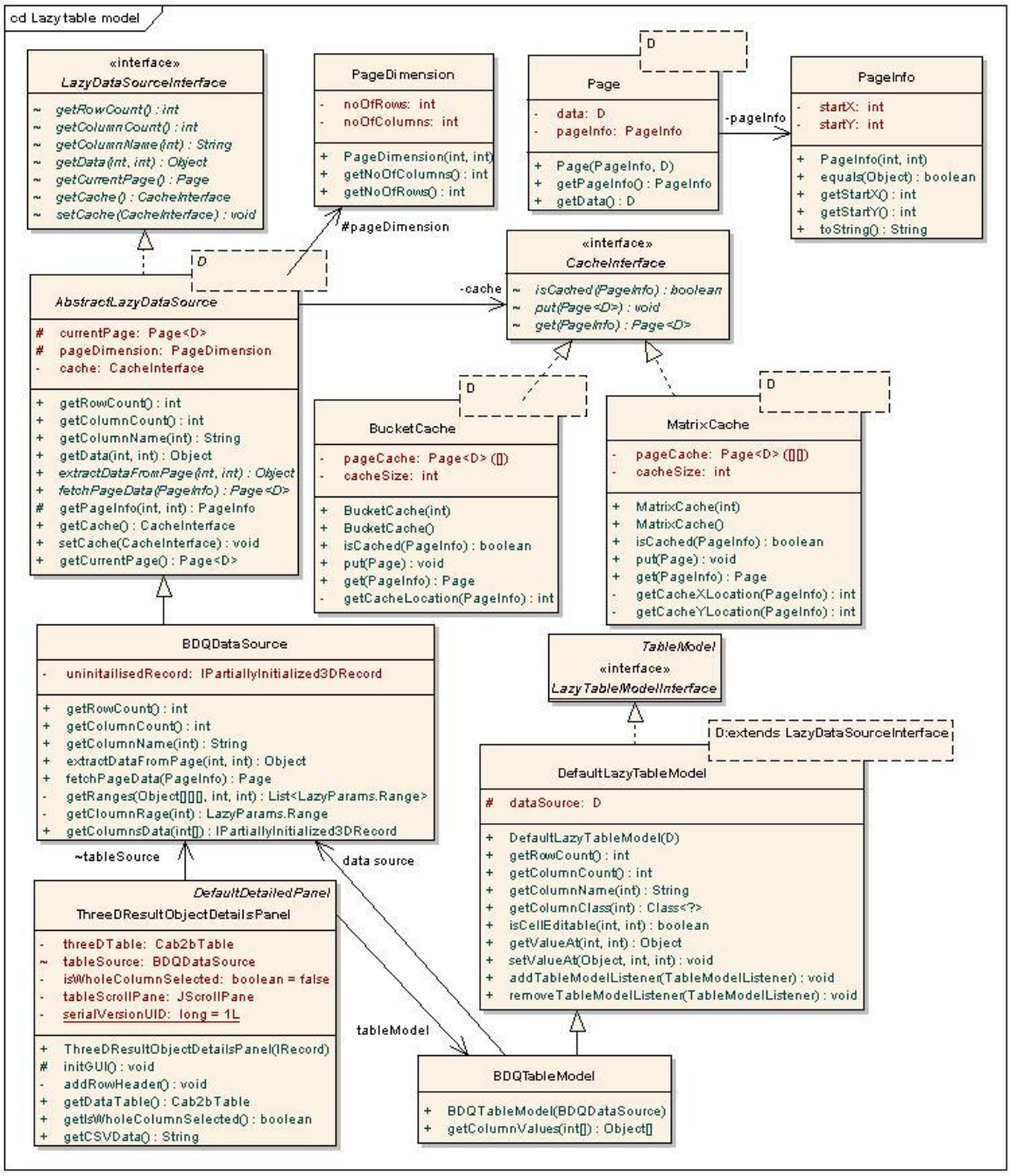

Figure 19 Classes Involved in Lazy Table Model component

This component has been used to display Bio Data cube object. Following sequence diagram shows further details.

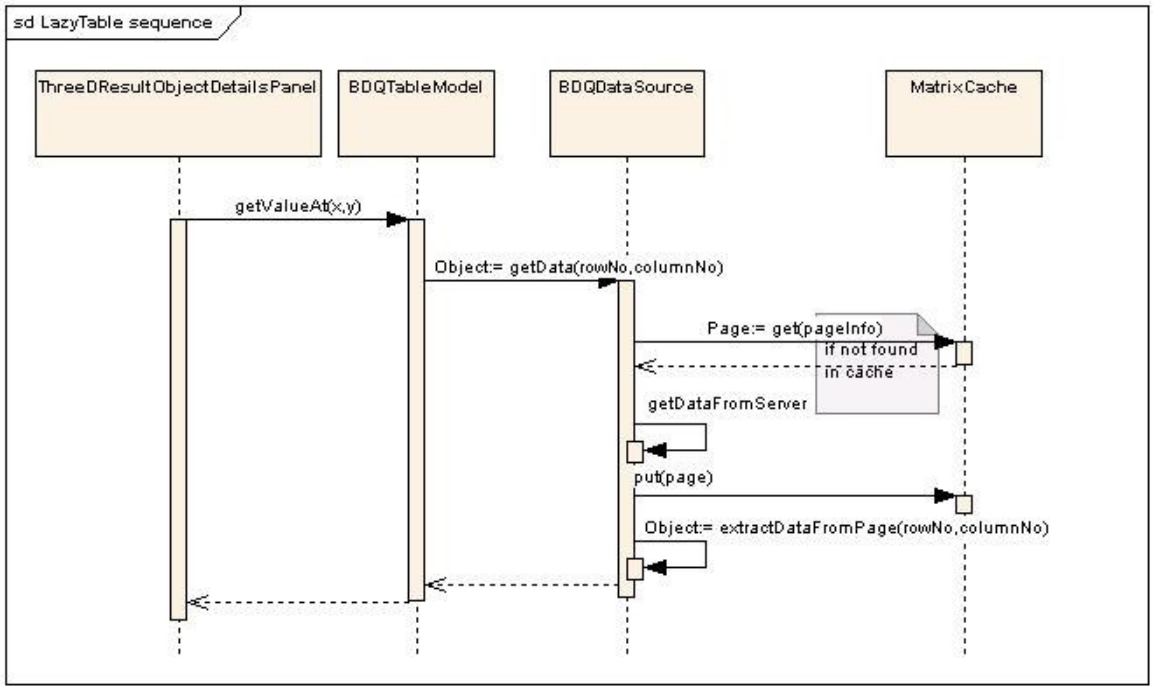

Figure 20 Flow of events in displaying BDQ

8 Dynamic UI generation for add/edit limits

8.1 Overview

The 'Add Limit' / 'Edit Limit' functionality of search data module allows the user to specify rules/constraints on the attributes of a selected entity like *"edu.wustl.fe.Gene"* or *"Gene Annotation"* respectively. Once the user decides the category on which to add a limit, the system auto generates the user interface with following properties:

- Alphabetically sorted list of attributes
- Attributes name are modified to make them user friendly manner (for example, clinicalDiagnosis should be Clinical Diagnosis)
- Based on the data type of each attribute
- Applicable set of operators are visible
- Data type based validation
- If the attribute has permissible values, these are displayed in a multi-select list box.

The section below describes the design details of dynamic UI generation for the Add / Edit limit functionality

8.2 Design

The dynamic UI generation is based on the following principles:

- The metadata for each attribute contains all the required information like data type and permissible values
- An XML file contains information about the display names for operators and UI properties.

For each entity the UI is auto generated based on its metadata and the XML file configuration

8.2.1 Metadata representation

Before we go into details of dynamic UI generation, is it important to understand how metadata for an entity is represented. For more details on these classes please refer to section Metadata Repository.

8.2.2 Dynamic UI configuration XML

This requirement needs mapping of attribute data type to all the information needed to visually render the UI component corresponding to that attribute. The information for rendering includes the following:

- List of operators for a given data type and context (enumerated or non-enumerated)
- The class name for the actual UI component to be instantiated, again for a given type and context.
- The UI component representing any attribute should show user friendly attribute name (i.e. by parsing the camel case words)
- Condition selection drop-down box
- Control to specify values for selected condition. This portion of the component is variable and changes according to the data type and context of the attribute (e.g. all attributes that contain enumeration, this would always be shown as a multi-select list box)

This mapping is captured in a configuration file in XML format. Reasons for the XML configuration file.

- Defining a configuration file to capture the mapping information helps abstract that information out of the code. This means some of the UI rendering information captured in the configuration file can change without having to compile the code.
- XML allows for validations by defining a DTD. The validation can further be made strict by defining data as actual xml elements. Thus (See Figure 3.0), the DTD mandates that the XML document have a data element for all the data types and include an operator list for all of them.

```
<!ELEMENT data-type-control (enumerated, non-enumerated)>
<!ELEMENT enumerated (string, number, boolean)>
<!ELEMENT string (conditions components)>
<!ELEMENT conditions (in, notin)>
<1ELEMENT in, notion, equals (displayname)>
```

Figure 21 Snippet of DTD used for dynamic UI configuration XML

data-type-control - This is the root node of the control and can contain enumerated or non-enumerated nodes as children
enumerated - This tag is parent of all the enumerated data types.
non-enumerated - This tag is parent of all the non-enumerated data type
string, number, boolean - these tags are the actual data type nodes which contain information such as operators associated with this node, display names of these operators, and the component which will render the attribute of this data type. Refer to file dynamicUI.xml for this.

8.2.3 Auto generation of UI

The configuration XML file is parsed using a DOM parser and the information is organized into the maps shown in the table below. This is a one-time activity and happens for the first instance, when UI needs to be dynamically generated for a class or category. All the logic is encapsulated in the **ParseXMLFile** class.

Map type	Details
Enum -Operator map **Note**: 'Enum' in this column implies an attribute containing permissible values.	**Key** = Enumeration representing data type. **Value** = Collection of enumerations representing operators
Enum-Component map	**Key** = Enumeration representing data type. **Value** = Name of UI component to be rendered
Non-Enum-Operator map	**Key** = Enumeration representing data type. **Value** = Collection of enumerations representing operators
Non-Enumerated-Component map	**Key** = Enumeration representing data type. **Value** = Name of UI component to be rendered

Given an 'EntityInterface', for every 'AttributeInterface' contained therein, the dynamic UI generation generates the UI component (details given here), based on the metadata of the attribute. The following flow-chart explains details for this activity:

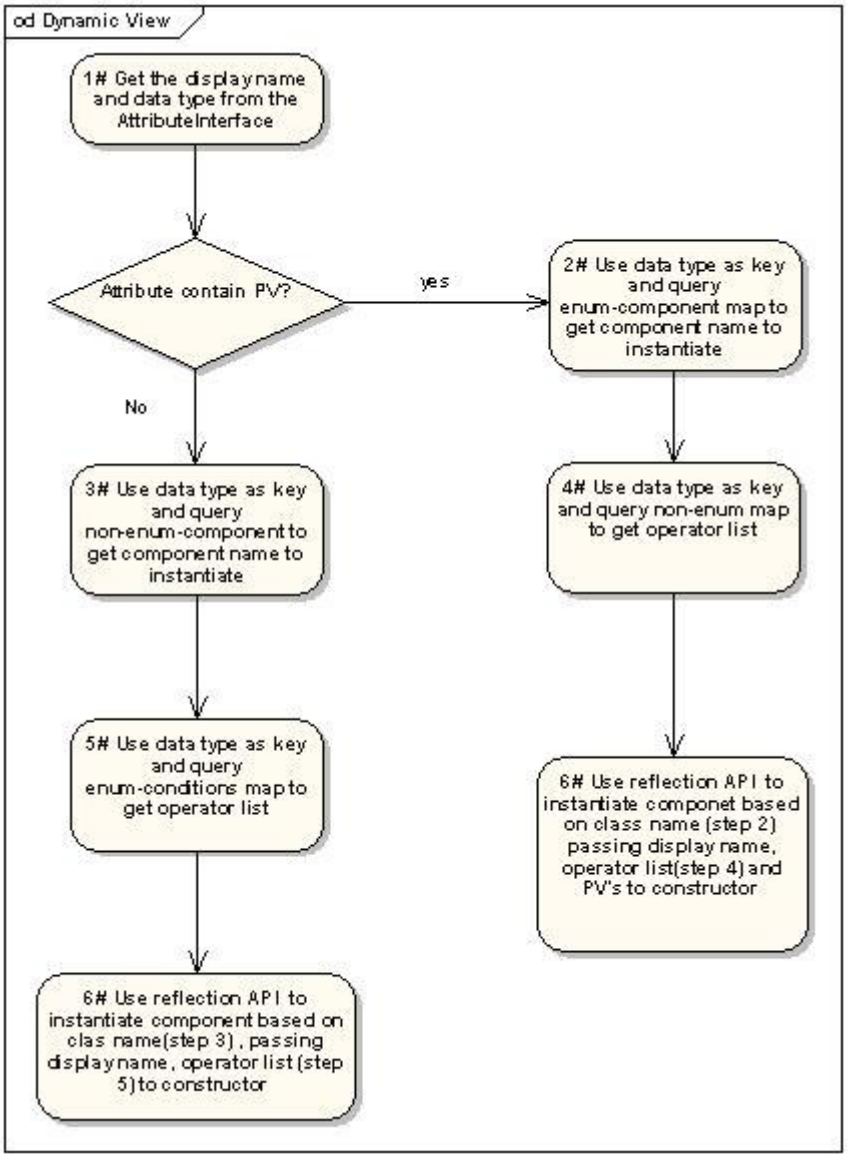

Figure 22 Detailed steps for generating UI component for an attribute

The **SwingUIManager** class has a static method *generateUIPanel(EntityInterface)* that iterates over the collection of attributes and processes metadata information based on the flow chart above to generate the UI component (**Cab2bPanel**) for that attribute. It then returns an array of these UI components that are added to a panel to represent the Add/Limit UI screen.

The UML diagram below shows the different classes involved in dynamic UI generation

IComponent - UI component should provide API to get selected condition, corresponding values and the attribute entity it represents. Thus one needs to have a common interface containing these APIs, which every data type specific UI component should implement. This is the interface containing methods to get/set UI component details for every attribute type.

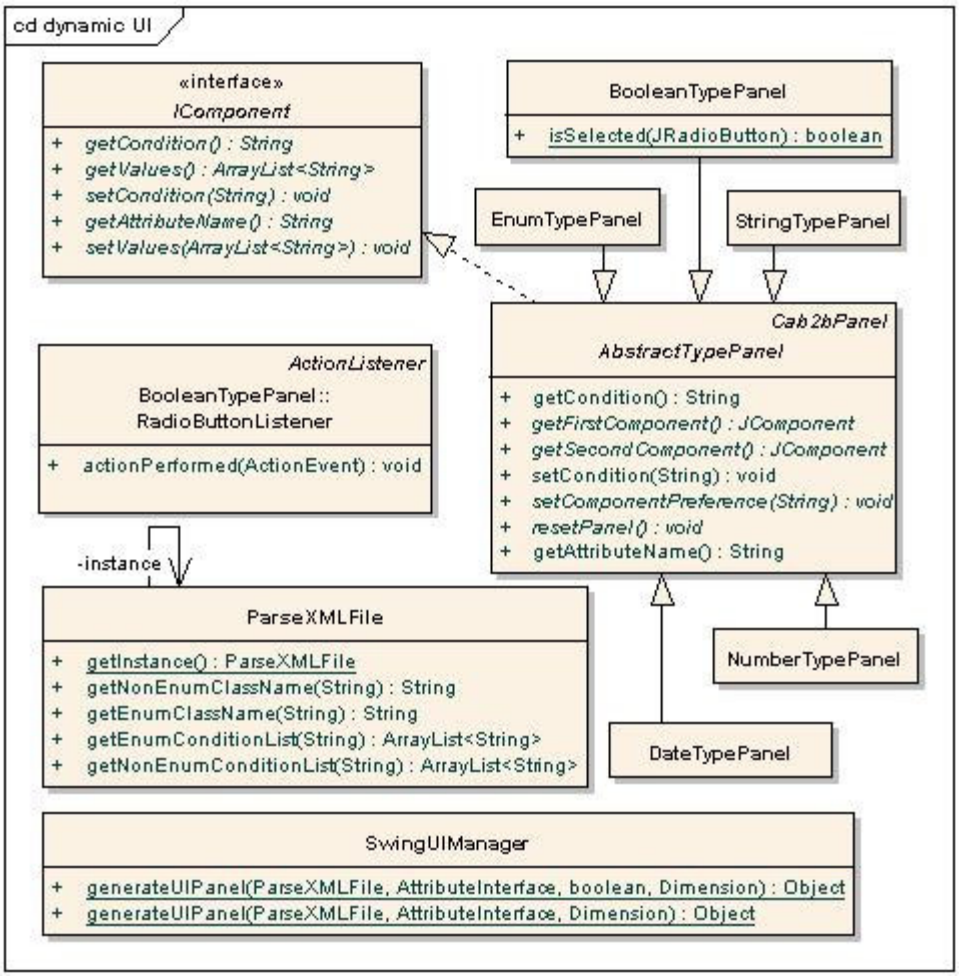

Figure 23 Class diagram for classes participating in dynamic UI generation

AbstarctTypePanel - This is an abstract UI component class, which contains common functionalities needed by all the attribute type UI components. It implements the IComponent. This component contains APIs to set UI for the condition list and user-friendly attribute names. Additionally it has abstract methods *getFirstComponent() and getSecondComponent()* to facilitate implementing class to provide the specific JComponent object specific to the specific data type. For each data type there is one class which extends this class, for example NumberTypePane for Numeric data types like integer, long, and double. StringTypePanel for String data type

9 Visual Query Interface OR Diagrammatic (DAG) view

9.1 Overview

The primary goal of this feature is to allow the user to do the following:

- View the category constraints added to the query in the form of graph nodes.
- Link the selected category constraints visually
- Edit / delete query constraints
- Resolve ambiguities if multiple paths are available between the source and target class / category constraints to link.
- The textual representation of the query expression

The basic design of the DAG view is to visually represent each constraint (i.e. a limit on class or category) as a node of the graph and allow linking of constraints as edges of the graph. NetBean's Graph Library supports visualization and editing of node-edge structures using drag and drop (org-netbeans-graph.jar), and it is platform independent.

This section describes the design for the same. Chapters Metadata search and Query Object are perquisites for this chapter.

The diagram (Figure 24 Basic workflow in the DAG) shows the basic workflow of the DAG view. The sequence of steps involved in the DAG view is as below:

1. User searches for the classes / categories for which he wants to form a query.
2. From the returned results, the user selects the class / category of interest. The Add Limit page shows all the attributes associated with selected class / category.
3. User specifies constraints on attributes and adds this constrained entity to query graph.
4. User may search and add different constraints to the query by repeating steps 1-3
5. User may select any two constrained entities and link them using the 'Connect Nodes' button.
6. If multiple paths are available for selected nodes, user may select multiple paths to connect these entities.
7. User may repeat step 6, to connect different constrained entities in the query graph
8. User may edit constrained entity and change the rules / constraints added on the attributes of the entity.

NOTE: Two entity nodes can be linked only if adding the selected path doesn't form a cycle in the query graph.

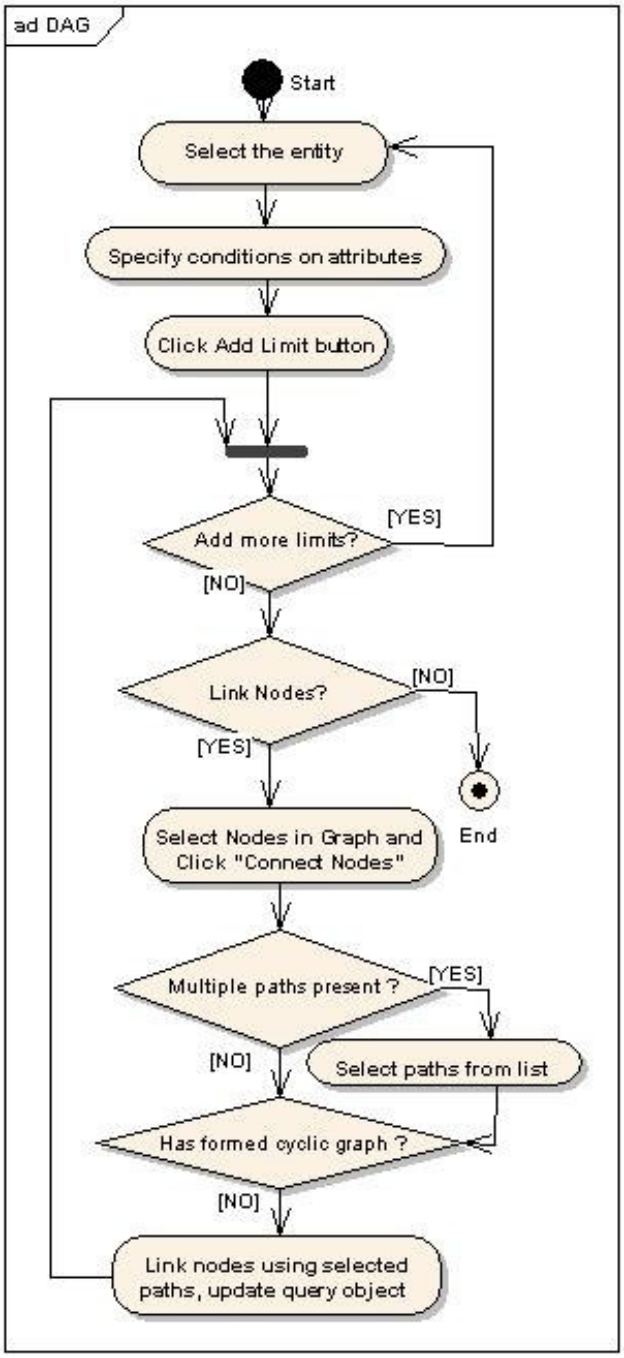

Figure 24 Basic workflow in the DAG

9.2 User Interface Design

This section describes the design of the user interface of the DAG view. It describes the classes that constitute visualization of the DAG view. The class diagram below details the classes and interactions amongst them.

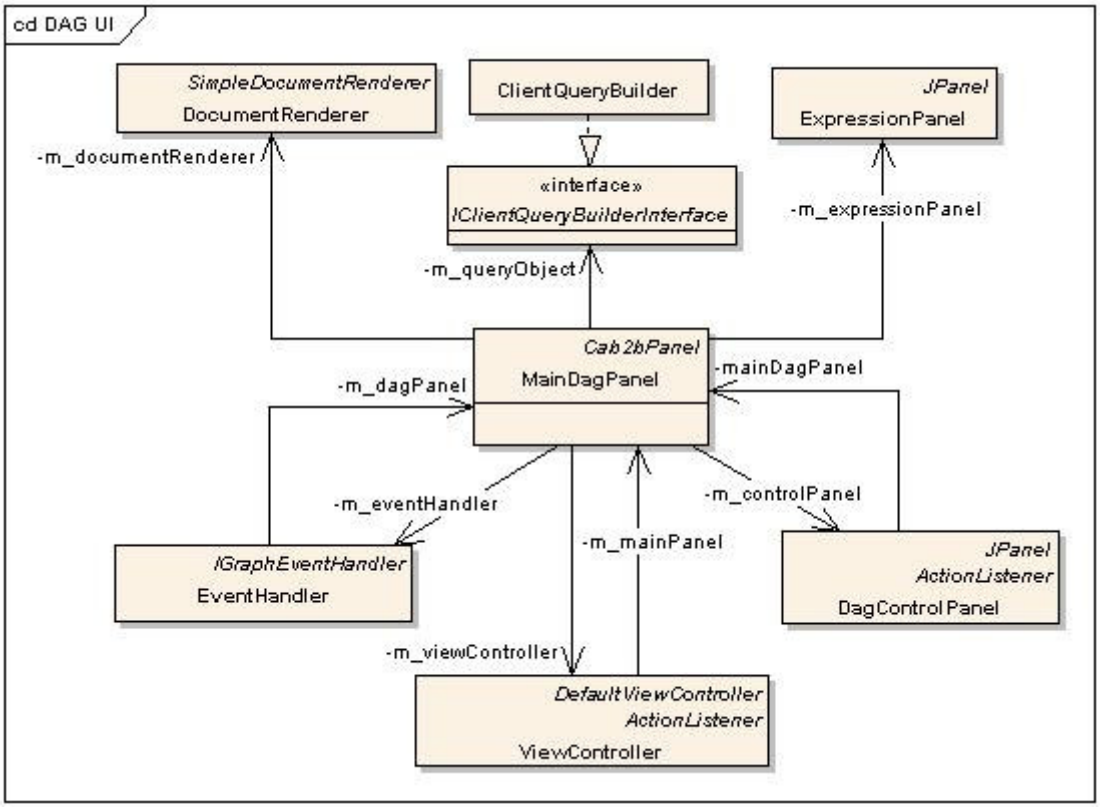

Figure 25 Class diagram for classes in the DAG view

- **MainDagPanel**: This class forms the core of the DAG view and is responsible for handling different user actions, related to query construction and updating the visual query graph. To display DAG on panel this class creates a JComponent that renders nodes and links using createView(…) method on GraphFactory from NetBean's graph library.Whenever a user adds a limit UpdateGraph(…) method adds an IExpression object representing the constrained category to the graph.
    - LinkNode(…) method links two nodes if the caB2B server has a valid path between the selected entities. If the system contains multiple paths between selected expressions, the ambiguity resolver allows the user to select paths of interest and links nodes with selected paths.
    - deletePath() and deleteExpression() methods delete the selected link and expression respectively from the UI as well as the backend query object. GetExpressionString () returns the textual representation of the IQuery object. This class also holds a reference of the IClientQueryBuilder (a wrapper over the IQuery). The backend query building section describes this in detail.
- **DagControlPanel:** This class controls different user activities such as liking selected nodes and clearing the DAG view.
- **ExpressionPanel**: This class provides the textual representation query object to the user.

Apart from these UI classes, there are classes, which hold UI details of every link and node that is rendered on the DAG panel. These classes and their details are described as follows:

- **GenericNode, ClassNode** and **ClassNodeRender** are involved in implementing the graph node functionality. These are the classes which hold information such as how to

render the node, what expression the node holds, and what other nodes are linked to this expression.

- **SimpleLinkRenderer** and **OrthogonalLinkRouterLinkRenderer** implement the graph link related functionality. These classes mainly perform rendering of graph link.
- **IconPortRenderer** and **SimplePortRenderer** are responsible for rendering connection ports. In order to link two nodes, one needs to add ports to the source and destination nodes.

Ambiguity resolver UI classes
This provides a user interface to show all the possible paths between selected source and target expression entities and allows the user to select paths of interest. This functionality queries the caB2B server to get all the available paths between source and destination expression entities. The class diagram below shows classes involved in implementing this functionality.

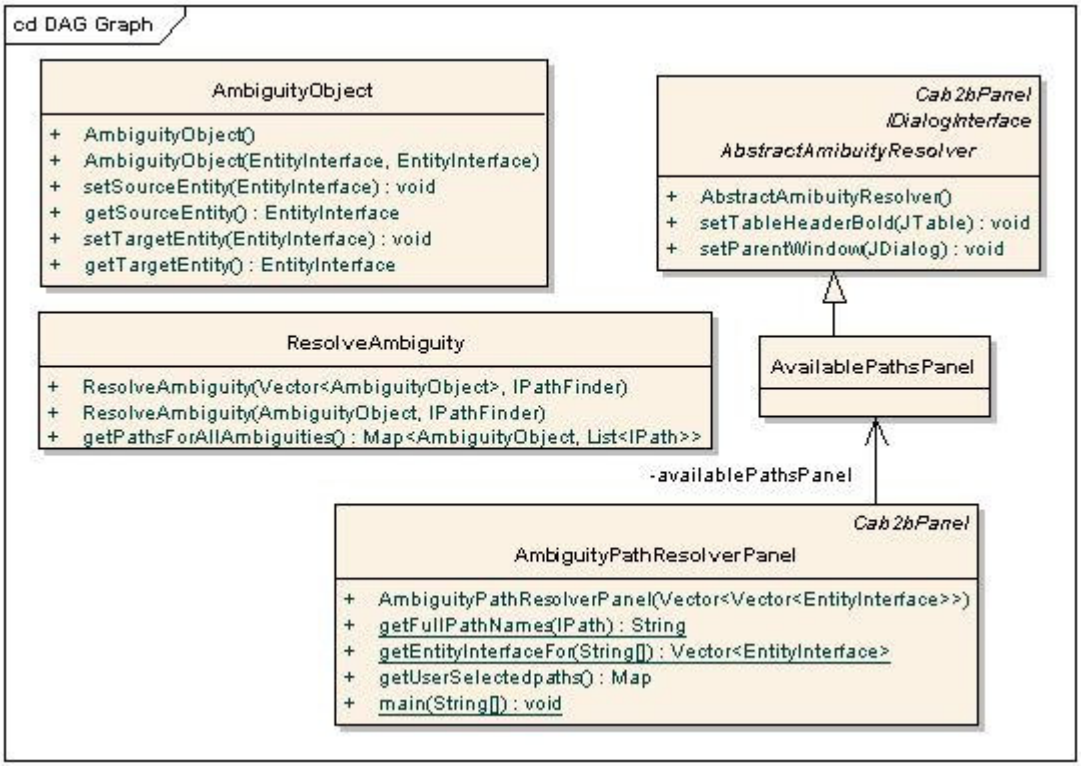

Figure 26 Class diagram for classes related to ambiguity resolver

- **ResolveAmbiguity**: This class queries caB2B server to get all the possible paths between source and destination entities and pops up a dialog box containing an instance of AvailablePathsPanel in order to allow the user to select multiple paths. In case of a single path, this class doesn't show this dialog box.
- **AvailablePathsPanel**: This panel holds the UI to show ambiguous paths.
- **AmbiguityObject**: The bean class holding the details of the entities between which the system has to find paths and resolve the ambiguity.
- **AmbiguityPathResolverPanel** This displays list of available paths for the current source, target entity and allows user to select one or more paths from it.

9.3 Query Building

The data of the visually constructed query is stored in an IQuery object. DAG holds a reference to this object. The IQuery object needs to be modified whenever the user links nodes, adds or

deletes links or nodes from the view. The **IClientQueryBuilder** interface defines method to update the query object according to the user's actions. **ClientQueryBuilder** implements the IClientQueryBuilder. The class diagram below shows different methods on interface.

addExpressio adds the constrained category element to query object when the user adds a limit. *addPath* adds the associations between source and destination entities specified by an *IPath* object. *removeExpression* method removes the expression with the specified ExpressionId from the graph. *removeAssociation* removes specified association between two ExpressionIds.

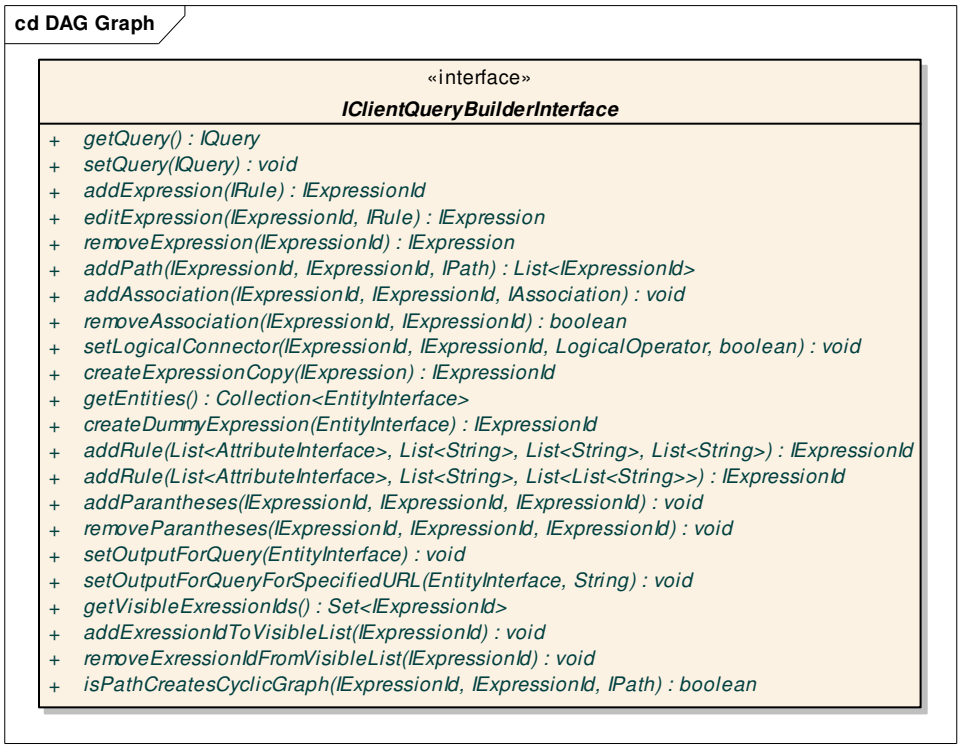

Figure 27 Client query builder interface for client side query building

10 Pagination Swing Component

10.1 Overview

In caB2B there are several instances where the user has to view large data sets. The examples of such instances include viewing results of a metadata search or viewing the results of a query. Traditionally, viewing of such large data sets is facilitated by enabling scrolling of the results. However, this approach makes it extremely cumbersome for the end-user to view the results, especially in case of larger data sets.

A better approach would be to paginate the results, much like the numbers of a book. This presents an organized view of the results and makes browsing large data sets extremely convenient.

Since it is required to show a paginated view at several places in the application, it becomes paramount to design a generic component (hence forth referred to as Pagination component) that can be re-used across all screens. In other words the nature of data to be paginated should not matter to the component.

In addition, it should be also being possible to configure the pagination component, both during initialization as well as dynamically, to paginate results based on some sorting criterion. The sorting criterion could vary from alphabetical to some context specific sorting (for e.g. sorting a category search based on the relevance of use). Thus the design for the generic component should be flexible enough to plug the different sorting algorithms, depending on the context in which it is used.

It should also be possible for the generic pagination component to select data elements for certain context-specific operations (like the ability to select data elements to add to the data list from the query results) by simply turning off or on the feature at the time of configuring and initializing the component in the application.

Given the generic nature of the component, the pagination component should dynamically compute the amount of space available on the screen and compute the number of elements to be displayed on a single page and consequently the total number of pages.

The pagination component is designed based on all the considerations mentioned above. Each element in the data set is displayed as a hyper-link with the provision to display some description associated with it. Additionally, the design allows for custom behavior for hyper-link clicks.

10.2 Design Details

The basic design for this component is based on the Swing UI MVC pattern; please refer to the UML Diagrams shown below.

10.2.1 View

The JPagination class constitutes the view for the component. It is an instance of JPanel and is further made up of the following components:

- **JGroupActionPanel** which extends JPanel and has hyperlinks which can perform group selection operation on the page elements. More is explained on group selection actions in Controller part.
- **PagePanel** which is a JPanel which is used to display the data elements for a given page. Each display element is again an instance of JPageElement which extends JPanel.

- **JPageBar** extends JPanel, contains hyperlinks to support navigation across pages.

The above three sub components can be arranged in any order.
The page elements which are displayed in page panel can be displayed in different configurations, the two important configurations are:

1. A linear list of page elements one below the other displayed in the page panel
2. A grid of page elements displayed in a matrix format. I.e. n page elements by m page elements.

Pagination component with three sub-component and check boxes for selections

Figure 28 Snapshot of a Pagination component

10.2.2 Controller

- **Selections**

Some of the group selection actions available on the page elements are:

1. Select All – by clicking on “Select All” hyperlink available in the group action panel, user can select all the visible and in-visible page elements.
2. Clear All – by clicking on the “Clear All” hyperlink available in the group action panel, user can clear all the selections they have made in the visible and in-visible pages.
3. Invert All – by clicking on the “Invert All” hyperlink available in the group action panel, user can invert the selections made in all visible and in-visible pages. Using Invert All action second time should bring back the original selections.

Group action panel can also be designed to have hyperlinks to select page elements that are in the visible page only, similarly to clear and invert the selections in the current or visible page only.

PageSelectionModel provides the backend for the above actions; there are other APIs in this class to get status on the current selections like

1. Number selections made till now, in all pages.
2. Is any element selected or not.
3. Is selection empty

JPagination provides API to add and remove **PageSelectionListener** to it. This enables user to listen for element selection, the event received is **PageSelectionEvent** if any page elements selection changes.

JPagination has API's to dynamically turn on and off the pagination component's elements selectable or not.

- **Navigation**

  There are basically three kinds of hyperlinks in the page bar to aide navigation

  1. "Next Page", "Previous Page" hyperlinks usually represented by ">", "<" characters are used to sequentially navigate forward or backward through pages. User click on these hyperlinks results in page panel showing the corresponding page.
  2. User can directly view any page by clicking a page index hyperlink. Page index hyperlinks will be numbers if the pager is numeric pager, alphabets if the pager is alphabetic pager, etc. These hyperlinks provide direct access to the desired page, unlike the Next Page", "Previous Page" hyperlinks which are for sequential access.
  3. Since there can be possibly many page index hyperlinks, page bar usually shows a small set of page index hyperlinks(5,10, etc) out of the all page indexes. So to provide navigation through these page index hyperlinks, there is "Next Page Indices" and "Previous Page Indices" hyperlinks. User action on these hyperlinks updates the current page indices hyperlinks that are visible.

  The text representing the "Next Page", "Previous Page", "Next Page Indices" and "Previous Page Indices" hyperlinks can be changed to any string or characters at the time of instantiation or dynamically(yet to implement).

Mouse Wheel Support

JPagination implements MouseWheelListener interface to provide fast sequential navigation through pages. Mouse wheel action automatically updates the current page index highlighting in the page bar.

Automatic Page Resize:

JPagination when provided with its parent components reference can automatically resize the pages(i.e. element per page) depending on the free space available with the parent component. This functionality is implemented by adding ComponentListener to the parent component and firing appropriate events when the parent component resizes in the action listener method

10.2.3 Model

The Pager interface and the implementing classes such as PaginationModel essentially form the data model. The Pager is an interface to the pagination model. The **AbstractPager** is an abstract class which provides the skeletal implementation for the Pager interface

AbstractPager also provides definition for final method subPage(). This method will be called only when the pager is non-numeric, to sub paginate the main pages whose size is more than desired. AbstractPager also keeps a copy of original collection of page elements intact, since this is needed for future use. Events like, changing the pager at runtime needs the original page elements collection (This functionality is not implemented in the current version).

AbstractPager has a map data structure which maps page index to a small collection of page elements called page. The Actually data structures are HashMap for map, String for page index and Vector for page. AbstractPager also keeps a ready list of all page indices that are there in the Map data structure as map's keys.

All concrete pager classes should extend the AbstractPager class and compulsorily override the page() method, with their own logic to paginate the elements except NumericPager.

The reason PaginationModel implements the Pager interface is to provide consistent API's to the view part. If PaginationModel doesn't conform to the Pager interface there can be chances where we introduce some methods in PaginationModel which are not there in Pager instances. The other way of think at it is, since Delegation in the patter used in PaginationModel it becomes a kind of norm to have all the methods which are available in Pager be present in PaginationModel.

Thus the pagination model and paginating process is clearly separated from the view part by using Pager interface and its concrete classes.
Pagination Levels: Pagination should be done at two levels

1) Level-1 Pagination: Can be any one of Numeric, Alphabetic, Keywords, Frequency, and etc based Pagination.
2) Level-2 Pagination: Is always a Numeric Pagination depending on the Level-1 Pagination. If Level-1 pagination is a Numeric Pagination then there is no need to have Level-2 Pagination. If Level-1 pagination is anything other than Numeric, we need Level-2 numeric pagination since non-numeric pagination doesn't conform to the condition that a page should have fixed number of page elements.

Thus level-1 pager is variable, it can be any kind of pagination, but level-2 pagination is always a numeric pager, if level-1 pager in non-numeric. And this probably explains the reason why the subPage method in AbstractPager is final, so that the actually concrete classes like AlphbeticPager can not override it, even by mistake.

If level-1 pager is a numeric pager then there is no work for sub page method.

Example: Let us consider Alphabetic pagination as the Level-1 pagination, there may be cases like page index "A" having a 20 page elements which can't be displayed on the screen without a scroll pane, but page index "B" may have only 2 page elements which will fit in one single page.

But for page with index "A" we have to again break the page with 20 elements into much smaller page. Numeric Pager is the best choice for this kind of Level-2 pagination.
Level-2 pagination depends on the page size of the selected page in the Level-1 pagination. Hence the page with index "A" the numeric pager may provide numeric page indices 1,2,3,4 for page with index "A", but for page with index "B" there is no need of second level page indices.
Note: Level-2 paging (sub paging) for non-numeric pager is not yet implemented.

10.2.4 UML Diagrams

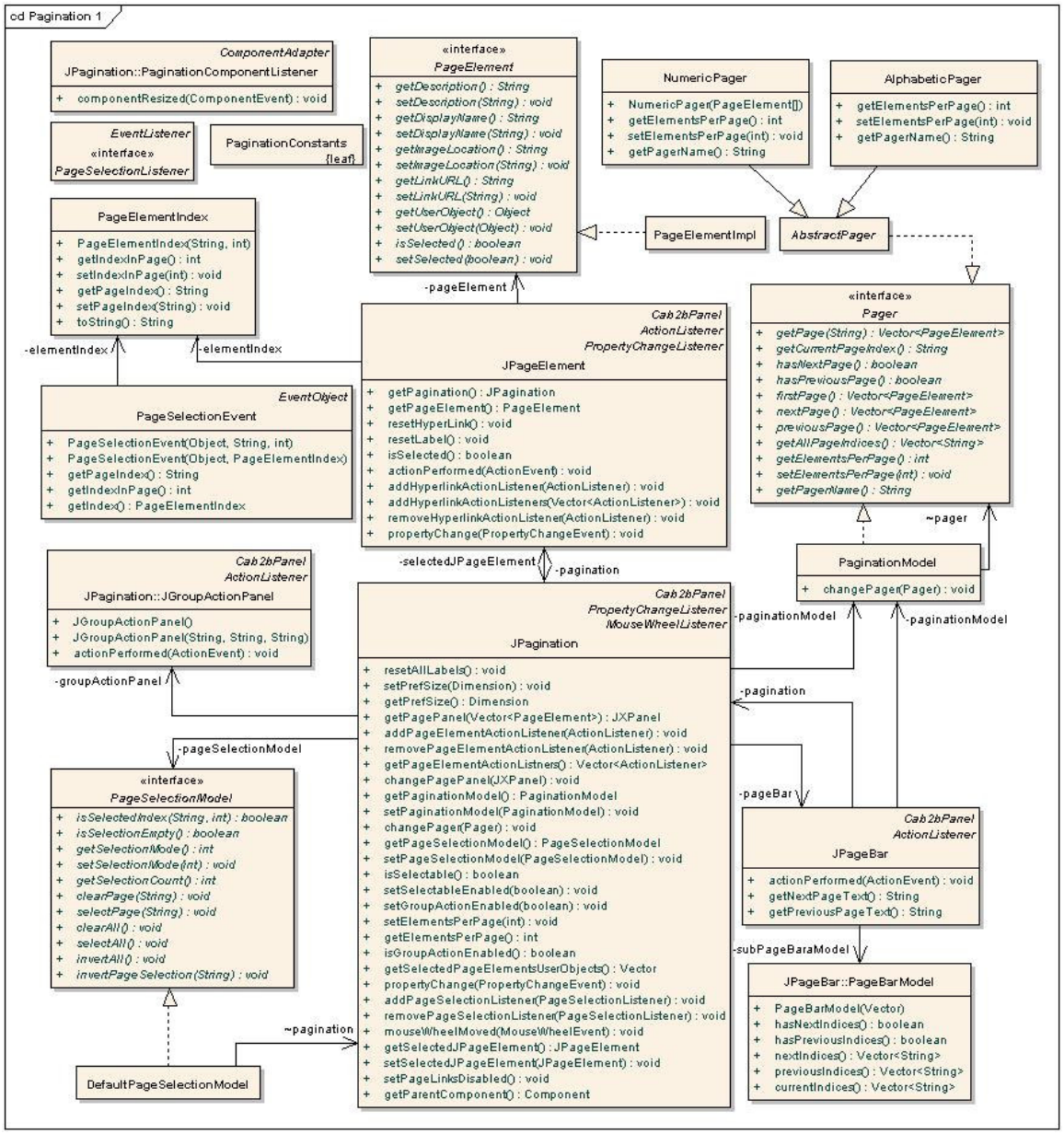

Figure 29 Classes involved in Pagination component

Sequence Diagram

The sequence diagram above describes the way in which pagination component is initialized first. The input to pagination component is a collection of page elements, each page element implementing PageElement interface

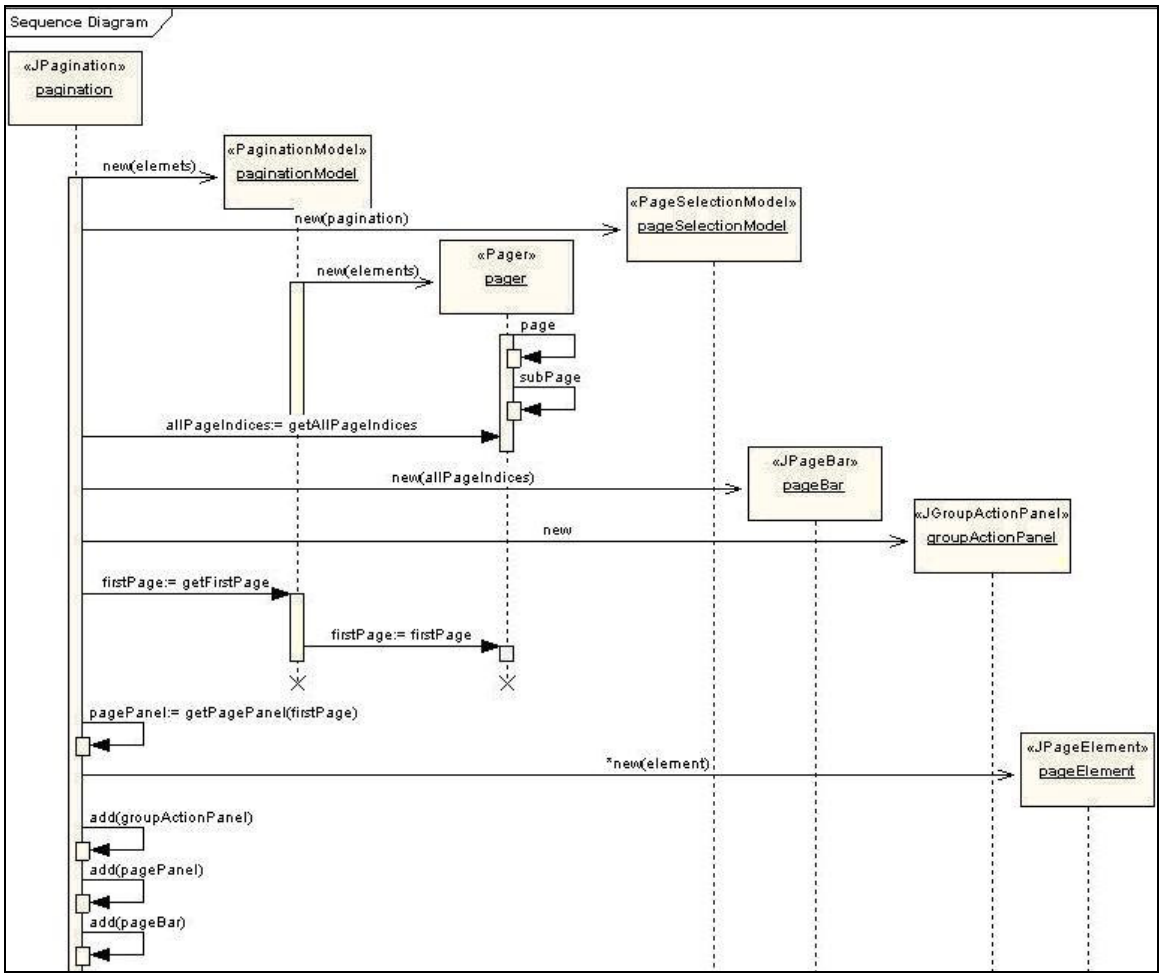

Figure 30 Pagination Sequence Diagram

The steps of event happening while constructing pagination component is explained as below:

1. JPagination accepts collection of elements as the parameter.
2. Creates a new instance of PaginationModel passing elements collection.
3. PaginationModel in-turn passes the elements collection to a subclass of Pager interface.
4. Pager internally calls the page() method to paginates the elements collection depending on some criteria.
5. JPagination then gets all page indices from the pager to construct JPageBar sub component.
6. JPagination gets first page from the pager to construct the page panel sub component.
7. JPagination constructs JGroupActionPanel sub component, and adds all these three subcomponents to it.

Pagination Usage in caB2B:

In the current version of caB2B, pagination component is used in two places

1. In the advanced search feature to show the search result. Here, selection of page elements is not needed; hence elements don't have check boxes and the group action panel. This is achieved by calling appropriate API's in the JPagination class.

2. In View search result feature to show the results got from data services. Here selection of page elements is important, since user would like to add the selected elements to the data list. Hence check boxes and group action panels are enabled by calling appropriate API's in the JPagination class.

These are the two instances which highlight the fact that Pagination component is generic in nature, which can be used in scenarios where long list of data needs to be displayed in the GUI.

11 Search Data Wizard

11.1 Overview

The search data for experiment dialog is a wizard based UI that allows the end-user to sequentially follow all the steps required to build, fire and view the results of a caB2B query. In this document, we shall understand the basic composition of the wizard as well as the navigation mechanism while moving from one step to another.

11.2 Class Diagram

The following is the class diagram that illustrates the composition of the search dialog wizard.

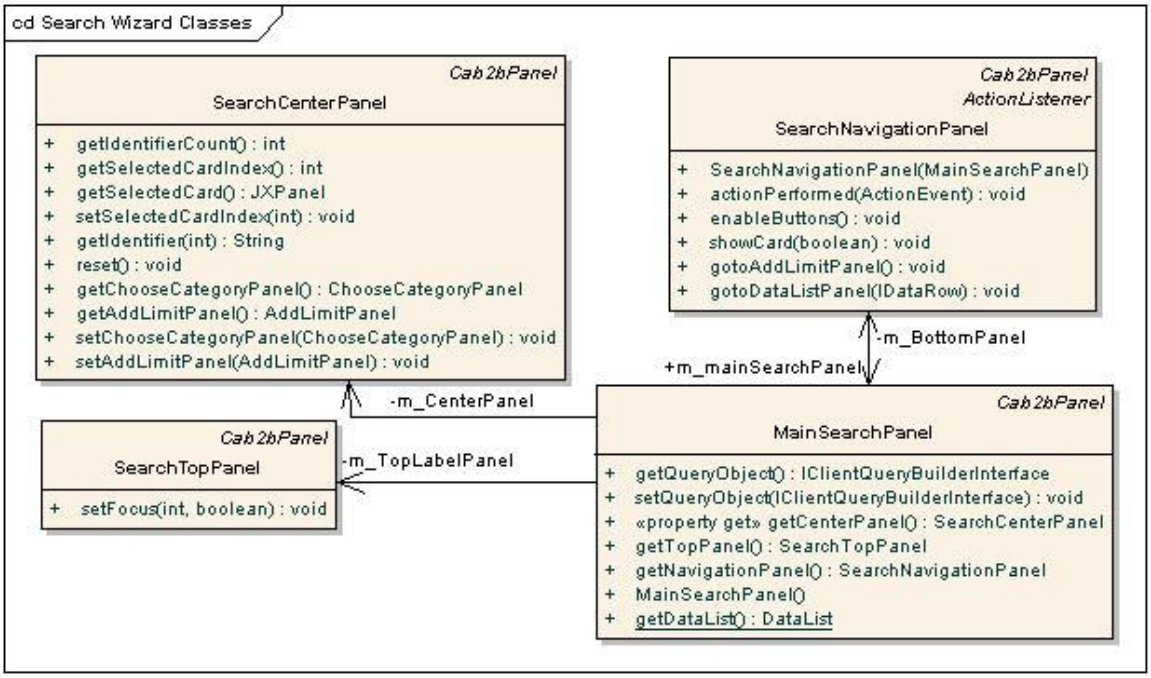

Figure 31 Class diagram for the Search dialog wizard

MainSearchPanel is the container class that represents the main wizard UI. It is an instance of *Cab2bPanel* and uses an instance of *BorderLayout* to manage the layout of its components. It is made up of the *SearchTopPanel* (added to the north region), the *SearchCenterPanel* (added to the center region) and the *SearchNavigationPanel* (added to the south region). The component is initialized at creation time.

The class provides getter methods to access each of these panels, so as to facilitate communication between the panels (For e.g. it is required for the *SearchNavigationPanel* to communicate with the *SearchTopPanel).* It also stores a reference to a cab2b implementation of the *IclientQueryBuilderInterface*, so that the reference can be available at every stage of the query building.

Cab2bPanel is a customized panel so that certain properties (like background color) can be centrally set and used across the application.

SearchTopPanel is the component that forms the top section of the wizard and its function is to visually indicate to the user the step that he is currently performing.

This component is an instance of *Cab2bPanel* and is composed of as many numbers of panels as there are steps in the wizard (in this case 5). It uses an instance of *GridLayout* to manage the layout of the child panels. Each panel is made of an instance of Cab2bLabel containing the appropriate text for the step in the wizard. The component is initialized at creation time such that panel corresponding to step1 has a white background and no border, while the panels for the remaining steps have a blue background and a *LineBorder*, which is black in color. The panel with the white background is always used to indicate to the end-user the step that he/she is currently performing.

The setFocus (int index, boolean blnForward) public API sets the background color of the panel corresponding to the step indicated by the index parameter, to white. The boolean parameter is used to indicate the traversal of the navigation so the adjacent panel (to the left or right depending on direction of traversal) can be reset.

SearchCenterPanel is an instance of *Cab2bPanel* and is a container class for all the UI components needed for each step in the wizard. The UI component for each step is again an instance *Cab2bPanel*; thereby making this a container of as many Cab2bPanels as there are steps.

It uses an instance of *CardLayout* to manage all the cards or in other words to manage all the Cab2bPanels needed at each step of the wizard. This component is initialized to contain and show the first card corresponding to the first step. Subsequent cards are added dynamically based on action taken in previous steps and shown as and when the user navigates across steps. The component also maintains state information like the currently selected card (the current step the user is on), and provided getter and setter methods for accessing and setting the value respectively.

SearchNavigationPanel is the component that provides functionality to navigate across the wizard. It is an instance of *Cab2bPanel* and uses an instance of *FlowLayout* to manage the layout of all its child components (instances of Cab2bButtons to facilitate navigation)

The component is also the event listener for its child buttons. For all navigations in the forward direction (refer to sequence diagram), it queries the *SearchCenterPanel* to get the current step. It then loads and adds the UI component corresponding to the next step to the *SearchCenterPanel*, if that is not already added.
However, there is an exception to the above action. In case of viewing search results, the UI component for viewing search results is always newly created and added.

For navigations in the reverse direction the component merely asks the *SearchCenterPanel* to show the previous card. If the next or previous component is successfully loaded and added, it then calls the *setFocus()* API on the *SearchTopPanel* .

11.3 *Sequence Diagram*

The sequence diagram below illustrates the flow of control when the user chooses to navigate from step1 to step2 (for the first time) in the wizard.

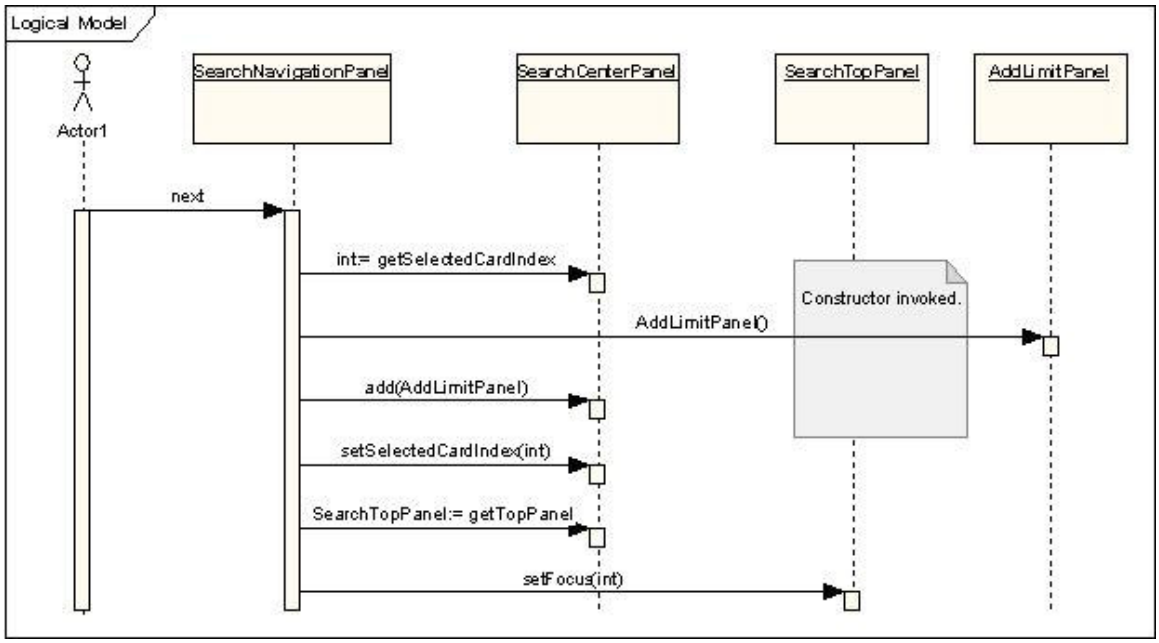

Figure 32 sequence diagram for navigation from step1 to step2 in the wizard

12 View Results

This section explains the classes involved in showing the query result.

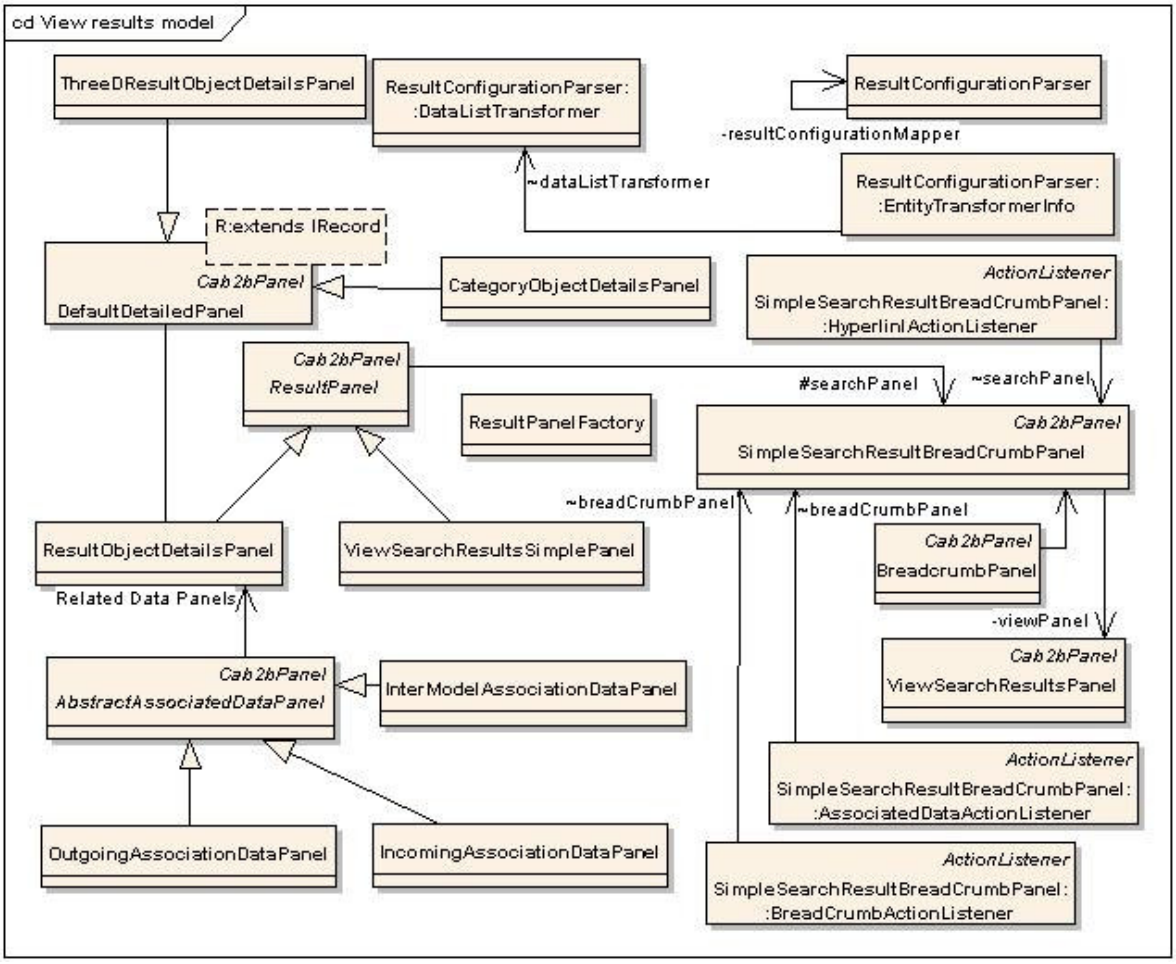

Figure 33 Classes involved in displaying query results

- **ViewSearchResultsPanel** displays the result of the query in two ways. In a simple view and a comprehensive view (not yet implemented).
- **SimpleSearchResultBreadCrumbPanel** represents one breadcrumb page of the result. It contains result panel, a data list summary panel and a breadcrumb panel to return to this result page. It contains result panel for a single object or list of object. It also provides the implantation for the listener of the breadcrumb, to view the details of a particular object and to show the related data.
- **ResultPanel** is an abstract class for the panel used to for the result. It provides the operation like add to data list and apply data list.
- **ViewSearchResultsSimplePanel** is a panel to display the list of objects. When the result of the query contains more than one object this panel is instantiated. It contains the pagination component to show the result.
- **ResultObjectDetailsPanel** is a panel to display the details of single object. When the result of the query contains a single object or when the detail of object is viewed, this panel is instantiated. It uses **ResultPanelFactory** to create the panel which is applicable

for the object to be shown. It also contains the reference to the applicable related data panels.

- **AbstractAssociatedDataPanel** is the base class for all the data panels used for showing associated (related) data. Its iniGUI() method creates the hyperlink for each of the related data.
- **IncomingAssociationDataPanel** represents the data which is related by incoming association for the object that is currently displayed.
- **OutgoingAssociationDataPanel**_represents the data which is related by outgoing association for the object that is currently displayed.
- **InterModelAssociationDataPanel** *r*epresents the data which is related by an inter model association for the object that is currently displayed.

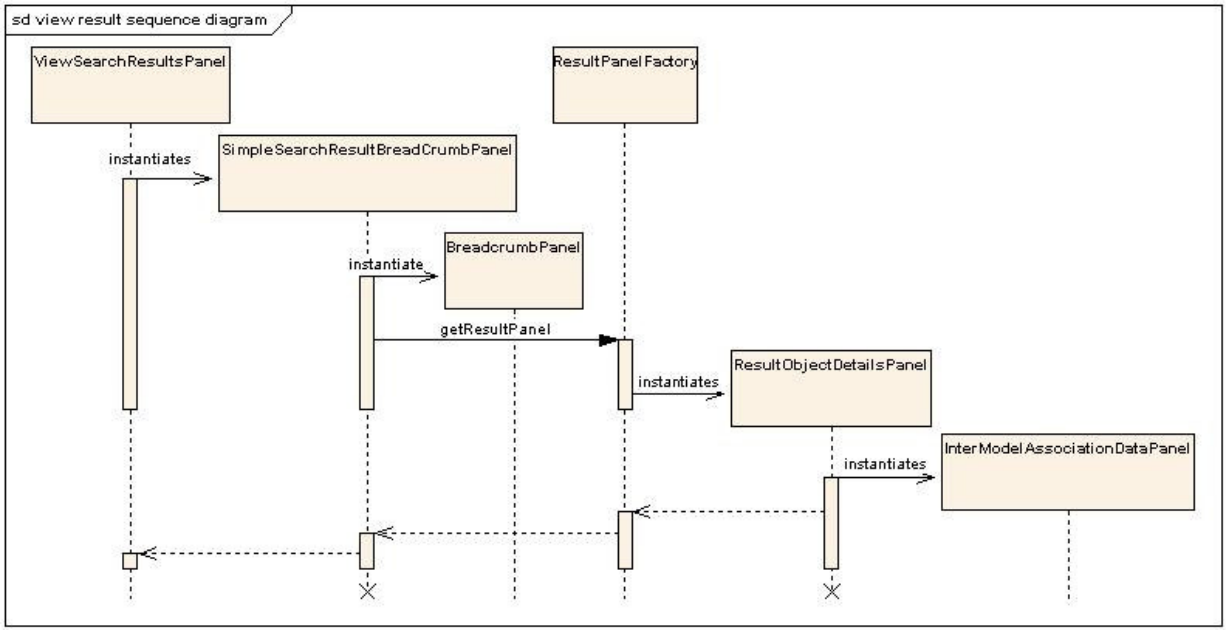

Figure 34 Order of instantiation of panels for view results

13 Record Customization

13.1 Overview

A user-defined query, represented by the **query object,** is transformed to appropriate DCQL. *DCQLQueryResults* obtained by executing this DCQL is then transformed into an *IQueryResult*. *IQueryResult* is a caB2B-specific representation of the results. Logically*, IQueryResult* is a collection of records (represented by *IRecord's)*. This chapter explains how this caB2B-specific representation (i.e. IRecord) can be customized based on the application/category being queried.

The *IRecord*
IRecord is a map from an attribute to its value.

13.2 Why customize IRecord?

The default *IRecord* represents the record of a UML class, as obtained from a data service that uses the default (de)serialization mechanisms of caGrid. A custom subtype of *IRecord* would be defined to add more information to such a record. Such a need can arise due to following reasons:

- **Custom (de)serialization by data services**
  A service might in some cases return more information than can be represented by *IRecord.* An example is the caArray service that returns identifiers of classes associated to the target class. To store this information, a custom record has to be defined.
- **Complex attributes**
  Some entities can have complex attributes which cannot be represented directly in *IRecord.* For example,
  - A *BioAssayData* record obtained from caArray data service has a *bioDataCube* attribute. This is a three-dimensional array of objects.
  - Each category record has other associated children category records.

13.3 Steps in customizing a record

1. Identify the entity or application for which the customized record has to be defined. Define appropriate subtype of *IRecord,* say *ICustomRecord.*
2. Implement any of the following components related to this customization:
   - Query result transformer: Responsible for transforming CQLResults into *ICustomRecord*.
   - Record details UI panel: Responsible for displaying *ICustomRecord* on the UI.
   - Data list transformers: Specify how an *ICustomRecord* is persisted as part of a datalist
     - Data list saver: Responsible for saving an *ICustomRecord* when it is part of a data list.
     - Data list retriever: Responsible for creating an appropriate *ICustomRecord* while retrieving a data list.
3. Register these implementations in the configuration xml "ResultConfiguration.xml".

13.4 Result Configuration XML

```
<applications>
    <application name="caArray" >
      <entity name="gov.nih.nci.mageom.domain.BioAssayData.BioAssayData">
        <result-transformer>
            cab2b.server.caarray.resulttransformer.BioAssayDataResultTransformer
        </result-transformer>
        <result-renderer>
            edu.wustl.cab2b.client.ui.viewresults.ThreeDResultObjectDetailsPanel
        </result-renderer>
        <data-list-transformers>
            <saver>cab2b.server.caarray.datalist.BioAssayDataDataListSaver</saver>
            <retriever>cab2b.server.caarray.datalist.BioAssayDataDataListRetriever</retriever>
        </data-list-transformers>
      </entity>

      <entity name="gov.nih.nci.mageom.domain.BioAssayData.DerivedBioAssayData">
        .
        .
      </entity>

      <default>
        <result-transformer>
          cab2b.server.caarray.resulttransformer.DefaultCaArrayResultTransformer
        </result-transformer>
      </default>
    </application>

    <default>
        <result-transformer>
            edu.wustl.cab2b.server.queryengine.resulttransformers.DefaultQueryResultTransformer
        </result-transformer>
        <result-renderer>
            edu.wustl.cab2b.client.ui.viewresults.DefaultDetailedPanel
        </result-renderer>
        <data-list-transformers>
          <saver>edu.wustl.cab2b.server.datalist.DefaultDataListSaver</saver>
          <retriever>edu.wustl.cab2b.server.datalist.DefaultDataListRetriever</retriever>
        </data-list-transformers>
    </default>
</applications>
```

Figure 35 Sample ResultConfiguration.xml

Note that the following are provided by caB2B:

- Customizations for “CategoryEntityGroup”
- caB2B defaults (the outermost default tag)

caArray is an example of a custom-extension. caArray has its own default query-result-transfomer that overrides the caB2B default query-result-transformer. But caArray needs customized result-renderers and datalist-transformers for the class BioAssayData.

13.4.1 ResultConfigurationParser

This is a singleton class which parses the ResultConfiguration.xml file and provides following methods for accessing the entries. If no entry is found for a given entity, the caB2B default is returned.

- *getResultRenderer(String applicationName, String entityName)*
- *getResultTransformer(String applicationName, String entityName)*
- *getDataListSaver(String applicationName, String entityName)*
- *getDataListRetriever(String applicationName, String entityName)*

13.5 IRecord and its extensions

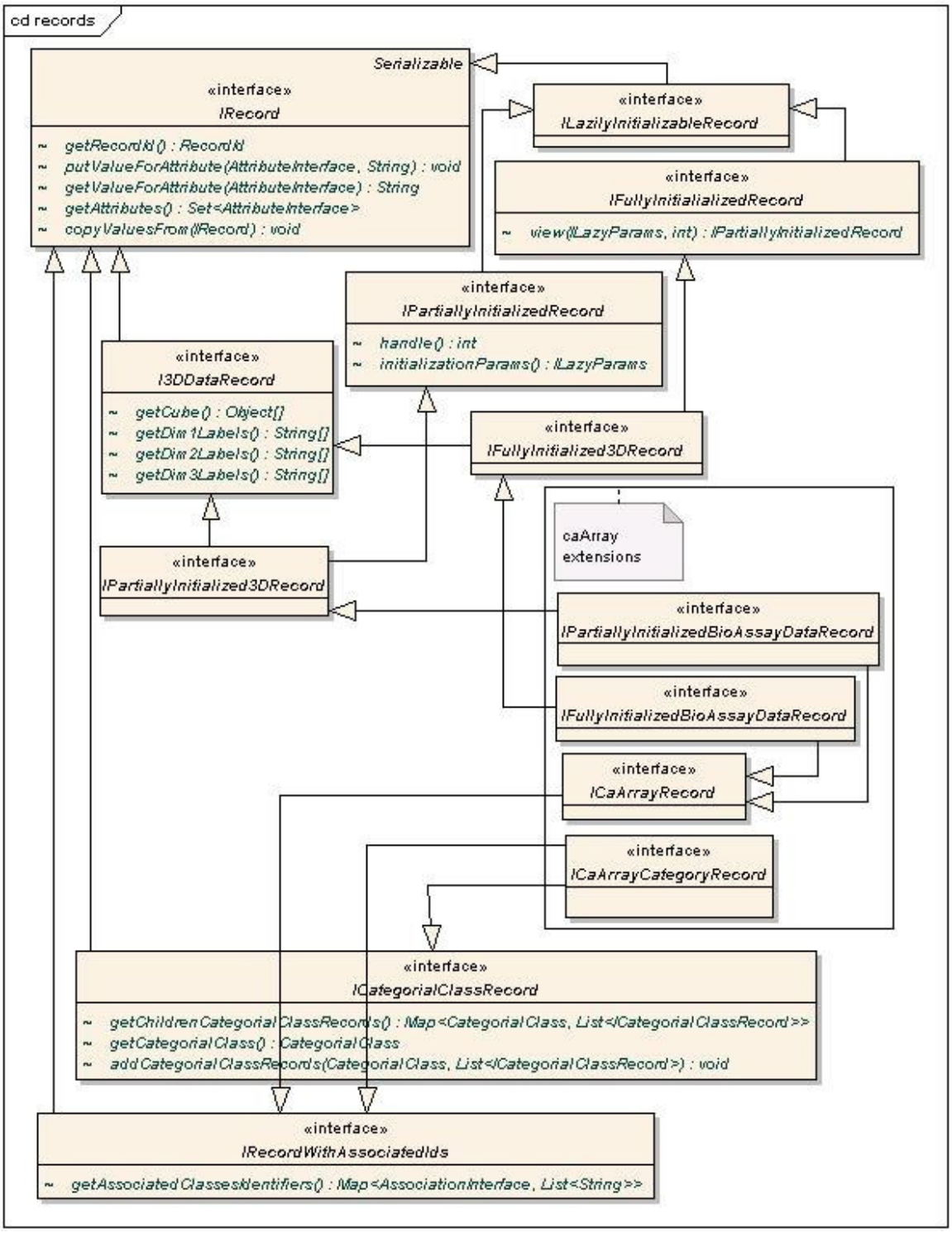

Figure 36 IRecord and its extensions

Following are the basic interfaces; other interfaces are either markers or mixins.

- **IRecord**: The most basic interface; it represents a record as a set of attribute-value pairs.

- **IRecordWithAssociatedIdentifiers**: Represents a record that can provide identifiers of associated classes as well.
- **I3DDataRecord**: Represents a record that has additional three-dimensional data. Methods provide the 3D matrix, and metadata about the dimensions.
- **IPartiallyInitializedRecord** and **IFullyInitializedRecord**: These interfaces are used for lazily initializing a record. See Lazy Table Model (Chapter **Custom UI components**) for more details.
- **ICategorialClassRecord**: Represents the records of a category. The records form a tree; the structure of the tree corresponds to the tree of classes in the category.

13.6 Query Result Transformers

A query result transformer is defined by the interface *edu.wustl.cab2b.server.queryengine.resulttransformers.IQueryResultTransfomer<R* ***extends*** *IRecord, C* ***extends*** *ICategorialClassRecord>* and is responsible for executing a DCQL and transforming the results into an appropriate *IQueryResult*. Following are the methods in *IQueryResultTransformer*:

- *IQueryResult<R> getResults(DCQLQuery query, EntityInterface targetEntity);*
  <R> the type of records created when executing a query for a class.

  Parameters:
  - **query** the DCQL.
  - **targetEntity** the target entity (corresponds to the target object of the dcql).
- *IQueryResult<C> getCategoryResults(DCQLQuery query, CategorialClass categorialClass);*
  <C> the type of records created when executing a query for a category.
  Parameters:
  - **query** the DCQL whose target object corresponds to the actual UML class represented by the categorial class.
  - **categorialClass** the categorial class.

Class diagrams for query result transformers are shown below.
Note: The text on the generalization links refers to type parameters e.g. declaration of DefaultQueryResultTransformer is ***class*** *DefaultQueryResultTransformer* ***extends*** *AbstractQueryResultTransformer<IRecord, ICategorialClassRecord>*
QueryResultTransformerFactory refers *ResultConfigurationParser* to obtain the appropriate transformer.

13.6.1 Inbuilt implementations of IQueryResultTransformer

- **AbstractQueryResultTransformer** This abstract class provides a skeletal implementation of a query result transformer. Concrete implementations need only implement the `createRecords()` and `createCategoryRecords()` methods. Additional hooks are provided and can be used to customize the creation and population of the records in the result.
- **DefaultQueryResultTransformer** This is the caB2B default query result transformer. It parses the *gov.nih.nci.cagrid.cqlresultset.CQLQueryResults* xml and extracts the values for the attributes of the target entity. The records in the results are of the basic types *IRecord* and *ICategorialClassRecord*.

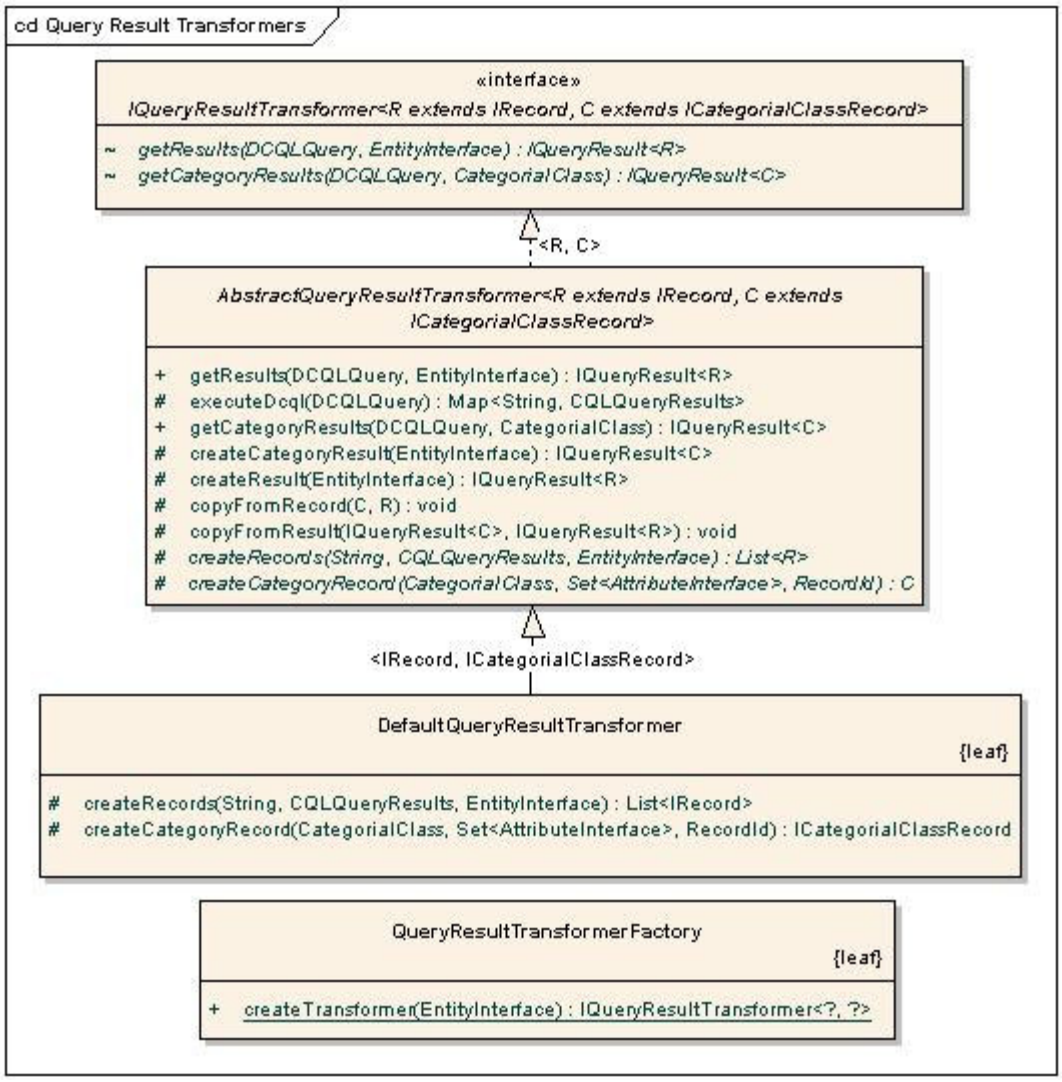

Figure 37 Query Result Transformers

13.6.2 Customization example – caArray

The interface *cab2b.server.caarray.resulttransformer.ICaArrayRecord* is used to represent a record of the caArray application. As explained previously, the caArray service returns identifiers of classes associated to the target class. Thus, an application-level transformer is defined for caArray that uses the caArray deserializers and reads this information.

- **AbstractCaArrayResultTransfomer:** Provides an implementation of the method *createRecords()* of *AbstractQueryResultTransformer.* It also handles the deserialization of the caArray results xml into objects and transforms these objects to *ICaarrayRecord* using reflection.
- **DefaultCaArrayResultTransformer:** This is the caArray application level default transformer.
- **BioAssayDataResultTransformer:** The caArray service returns a biodata cube as data associated to any *BioAssayData*. This transformer is required to read the biodatacube and transform it to an appropriate *IPartiallyInitializedBioAssayDataRecord.* (For details of lazy initialization, refer Lazy Table Model (Chapter **Custom UI components**)

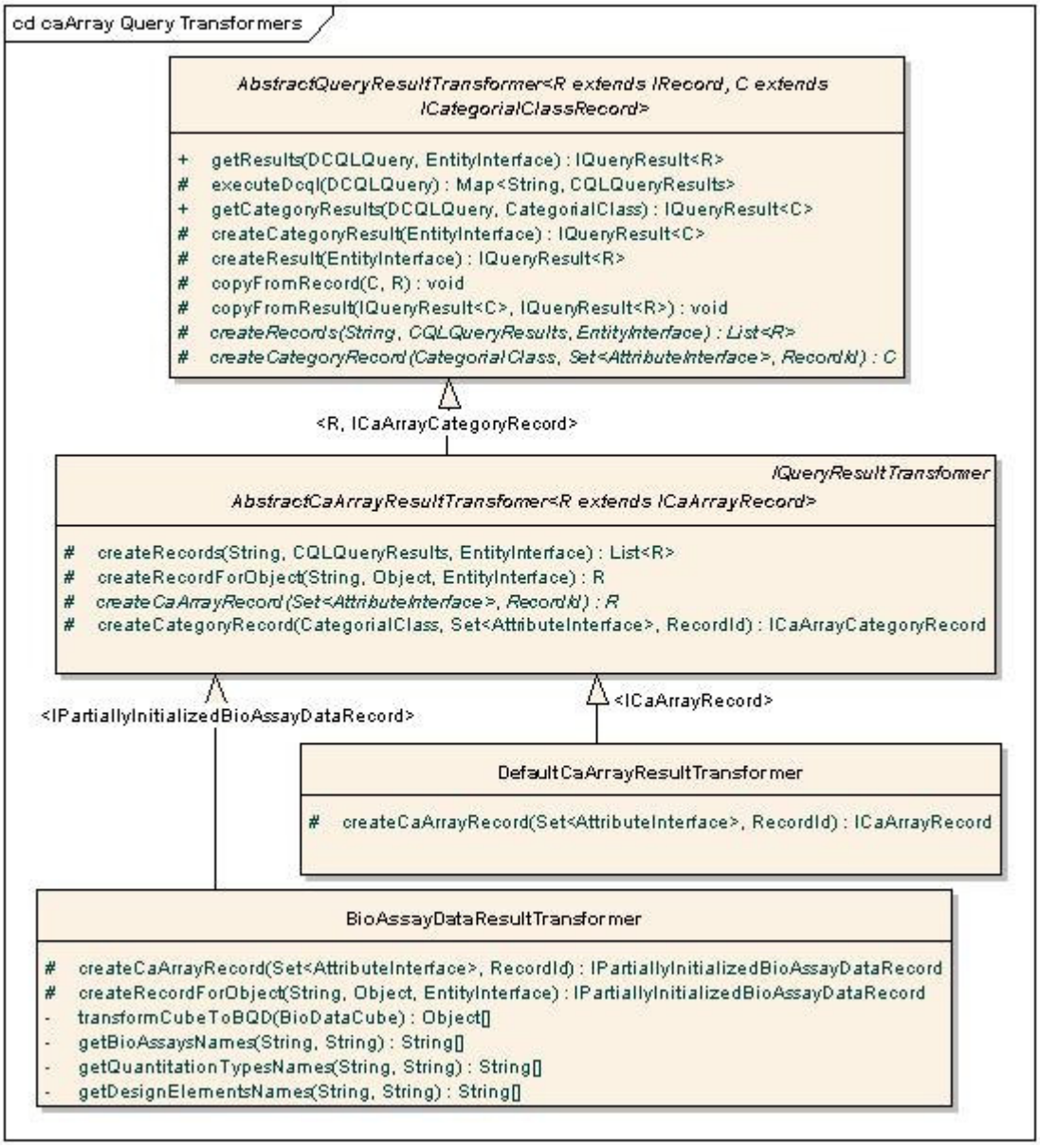

Figure 38 Query Result transformers

13.7 Data list transformers

A data list is saved using the dynamic extensions (DE) API. To do this, following transformations are needed:

- From *IRecord* to DE specific representation of the record; this is needed while saving a data list.
- From DE specific representation of a record to its corresponding *IRecord*; this is needed while retrieving a data list.

The corresponding *saver* and *retriever* interfaces are

- *edu.wustl.cab2b.server.datalist.DataListSaver*

- *edu.wustl.cab2b.server.datalist.DataListRetrieve*

These interfaces identify the operations that can vary depending on the record customization. For saving a data list, a new entity is created for which records are populated. A *saver* customizes the attributes/associations of the new entity that is created.

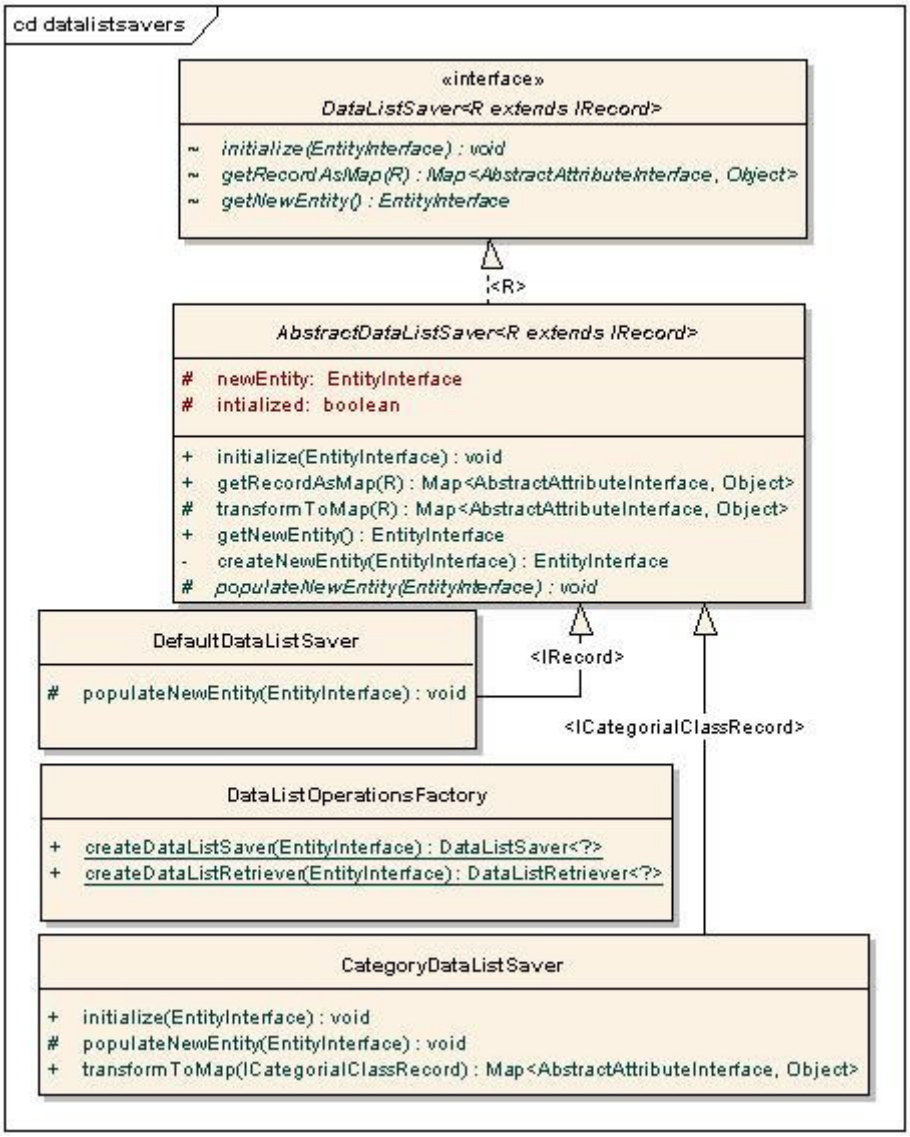

Figure 39 Data list savers and factory

For example, consider a specialization of *IRecord* called *IFooBarRecord* which represents records for an entity *FooBarEnt*. *IFooBarRecord* provides additional info, say, through the method *getFoo*(). In this case, we can have a *FooBarSaver* and *FooBarRetriever*. *FooBarSaver.getNewEntity*() method will return an entity that contains all attributes from *FooBarEnt*, and an additional attribute called "foo". The method *FooBarSaver.getRecordAsMap*() will appropriately put an entry into the map for the attribute "foo", by reading the value from *IFooBarRecord.getFoo*().

Then, while retrieving the records, the value of the attribute "foo" of the entity "FooBarEnt" will be set for the property `IFooBarRecord.foo` by the corresponding *retriever.* This would be done in the method *FooBarRetriver. getEntityRecords(List<Long> recordIds).*

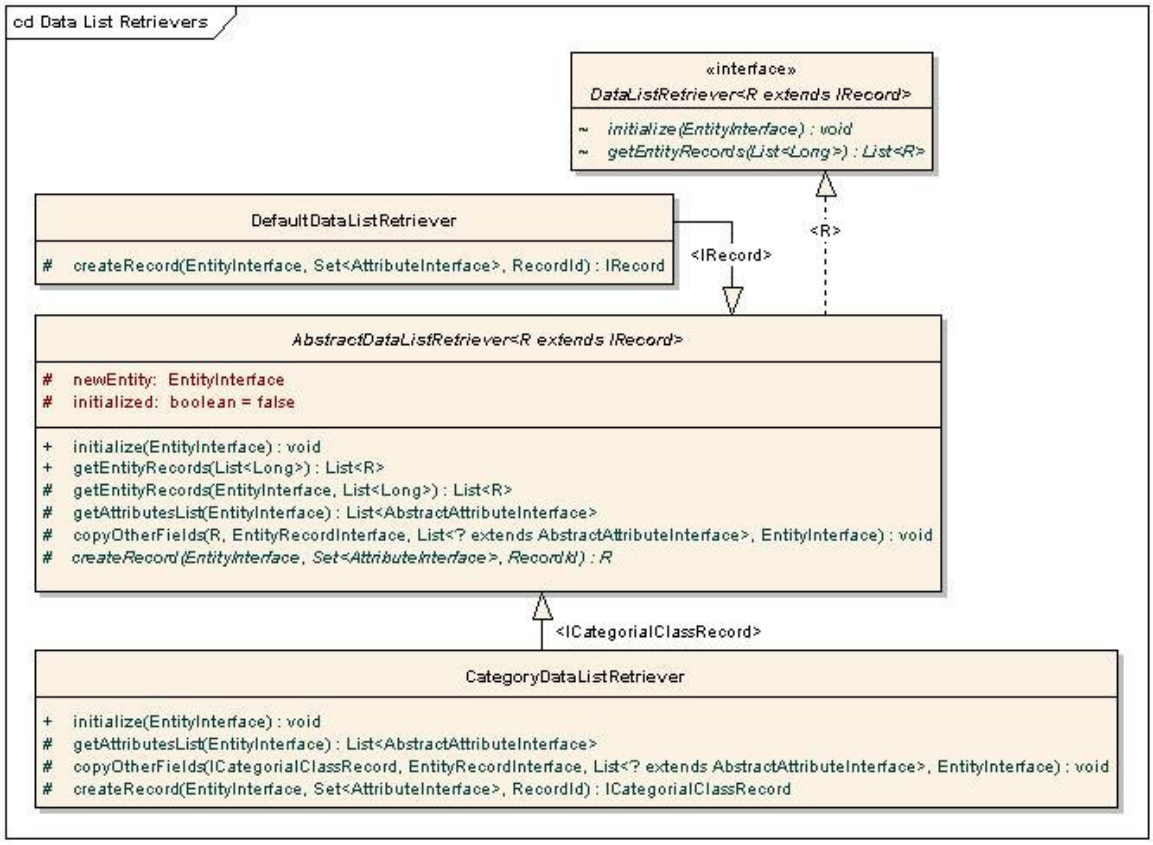

Figure 40 Data list retrievers

13.7.1 Inbuilt implementations of DataListSaver / DataListRetriever

- **AbstractDataListSaver:** Skeletal implementation of a DataListSaver. A concrete implementation need only implement the method populateNewEntity() to add attributes and/or associations to the newly created entity.
- **AbstractDataListRetriever:** Skeletal implementation of a DataListRetriever. A concrete implementation need only implement the method createRecord () to create an instance of appropriate subtype of *IRecord.*
- **DefaultDataListSaver:** This is the default caB2B data list saver; the new entity it creates is a clone of the original entity.
- **DefaultDataListRetriever:** This is the default caB2B data list retriever; it creates records of the basic type *IRecord.*
- **CategoryDataListSaver:** New entities are created to represent the classes and associations within the category and the records are stored into appropriate entities.
- **CategoryDataListRetriever:** The records from the multiple entities are grouped together to reconstruct the *ICategorialClassRecord*'s.

13.7.2 Customization example – caArray

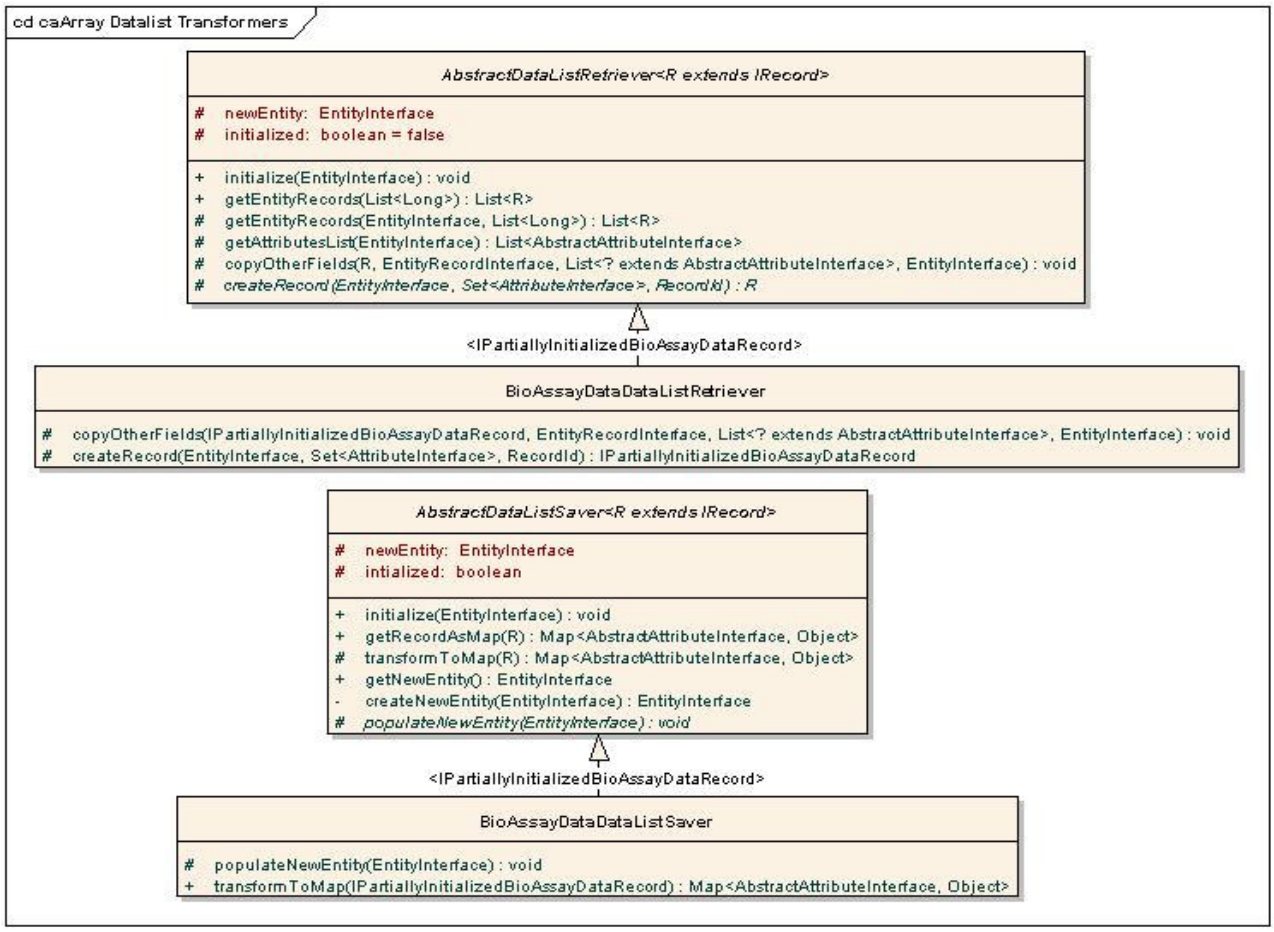

Figure 41 Caarray extensions for data list operations

- **BioAssayDataDataListSaver:** As explained previously, the records of *BioAssayData* contain a biodatacube which also has to be persisted. Currently, this saver creates blob columns for the biodatacube and its metadata.
- **BioAssayDataDataListRetriever:** This retriever reads the data from the corresponding blob columns and populates this in the *IPartiallyInitializedBioAssayDataRecord* representation of the record.

13.8 Result Renderers

The result renders are used to render the complete details of single record.

ResultPanelFactory uses **ResultConfigurationParser** to obtain the appropriate renderer for the given type of the record. The default render is e*du.wustl.cab2b.client.ui.viewresults.* **DefaultDetailedPanel**. It accepts IRecords and renders attributes and its values in the form of a table with a row for each attribute.

CategoryObjectDetailsPanel extends the functionality of default renderer to display the tree like structure of the category. It accepts **ICategorialClassRecord** as an input. It displays the attributes of the root categorical class using parent renderer (i.e. **DefaultDetailedPanel**). It creates **B2BTreeNode** structure for the associated child categories. If a particular child has a single record or one-one association with the parent its records are displayed along with the

parent itself and not in a separate node. **JTreeTable** is the custom UI component that accepts B2BTreeNode to display tree structure of the categories.

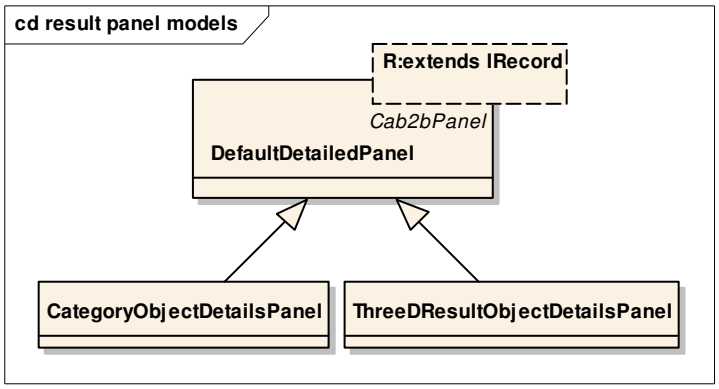

Figure 42 Result Panel Model

ThreeDResultObjectDetailsPanel *is* the renderer for the caArray object "Bio data cube". It is a three dimensional representation of micro array data. It uses **LazyTableModel** to show the huge data.

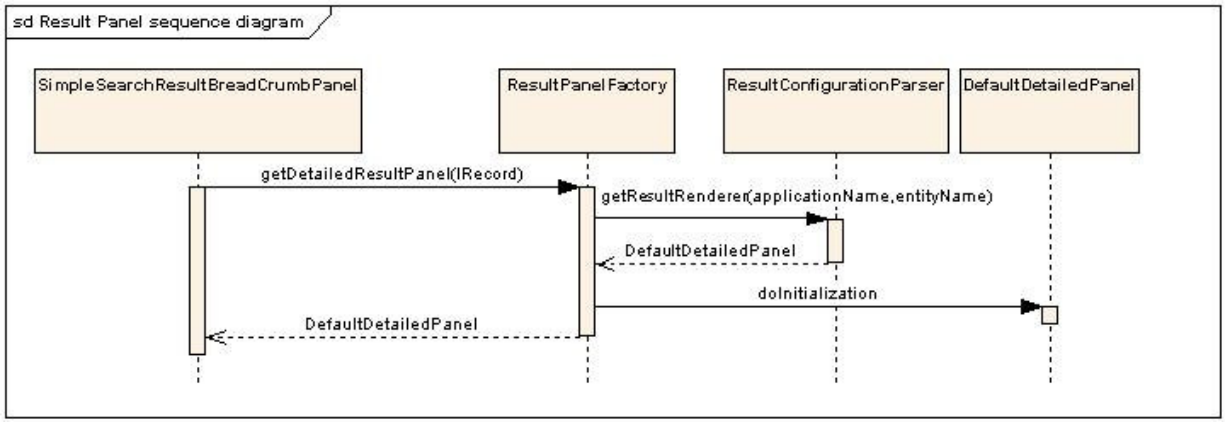

Figure 43 Flow of events while displaying results

14 Data List

14.1 Overview

The user's selected data is represented by the data list. After adding data into the data list, user can save it and create experiment out of it. It is more like a shopping cart where user adds the data in which he / she is interested.

14.2 View Data list

Following diagram illustrate the classes involved in displaying the data list.

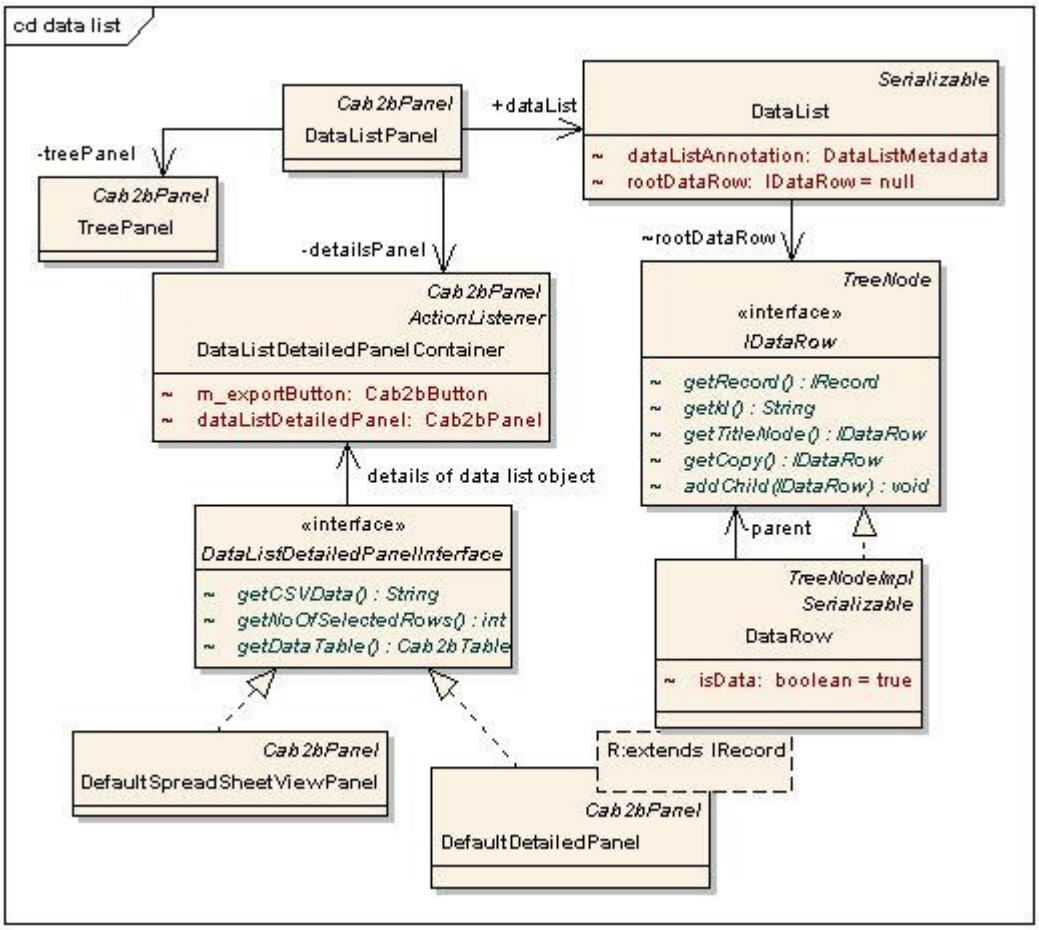

Figure 44 Classes involved in displaying data-list

- **DataRow** class represents a single object added into the data list. It gives the tree like structure if the user's data. The similar types of the objects are grouped by an IDataRow and it is distinguished by setting its *isData* flag to false (It is referred as a title node).
- **DataList** represents current selected data. It contains a tree of IDataRow. The root of the tree is represented by a single IDataRow.
- **DataListPanel** is the container panel for data list. It contains DataListDetailedPanelContainer and TreePanel
- **TreePanel** *contains* the tree of the data list. On selecting a particular node its details are displayed in the **DataListDetailedPanelContainer**.

- **DataListDetailedPanelInterface** is implemented by a class that can be used to show the details of objects in datalist. It getCSVData() method returns the comma separated values of the object. This is used for exporting the details of the object.
- **DataListDetailedPanelContainer** displays the details of the selected IDataRow from TreePanel. If it is title node, then objects under it are displayed in the form of the spreadsheet using DefaultSpreadSheetViewPanel. If it is a single object the details are displayed using DefaultDetailedPanel.

14.3 Data List Operations

Save and retrieve are the main operations related to data list.
A new model is created using the dynamic extensions (DE) API corresponding to each data list. Since a data list is a set of trees, a dummy root entity is created which has these trees as children. The dummy root entity is thus representative of the data list. Then the records are saved as records of the respective entities of this model using DE. These steps are orchestrated by the **DataListOperationsController**. It has two methods:

- DataListMetadata saveDataList(IDataRow rootRecordDataRow, DataListMetadata dataListMetadata). It saves the data list into the database. In the process, appropriate DataListSaver is invoked to obtain the DE specific representation of the records and the new model to be created. See Data list transformers (Chapter **CRecord Customization**)
- List<IRecord> getEntityRecords(Long entityId). Return records of the given entity. It delegates the operation to appropriate DataListRetriever. See Data list transformers (Chapter **CRecord Customization**)

Following are the sequence diagrams illustrating the flow while retrieving and saving records of a data list

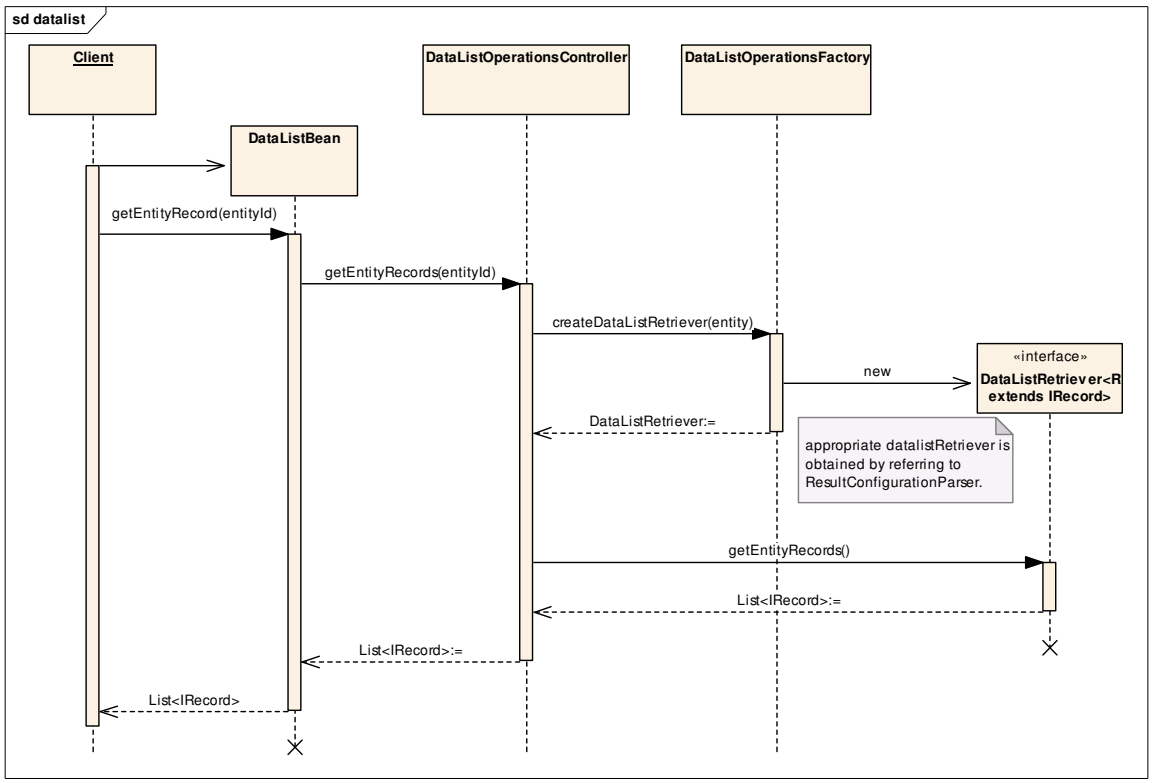

Figure 45 Sequence diagram for retrieving records of a data list

The factory DataListOperationsFactory provides the appropriate DataListRetriever or DataListSaver by referring to ResultConfigurationParser..

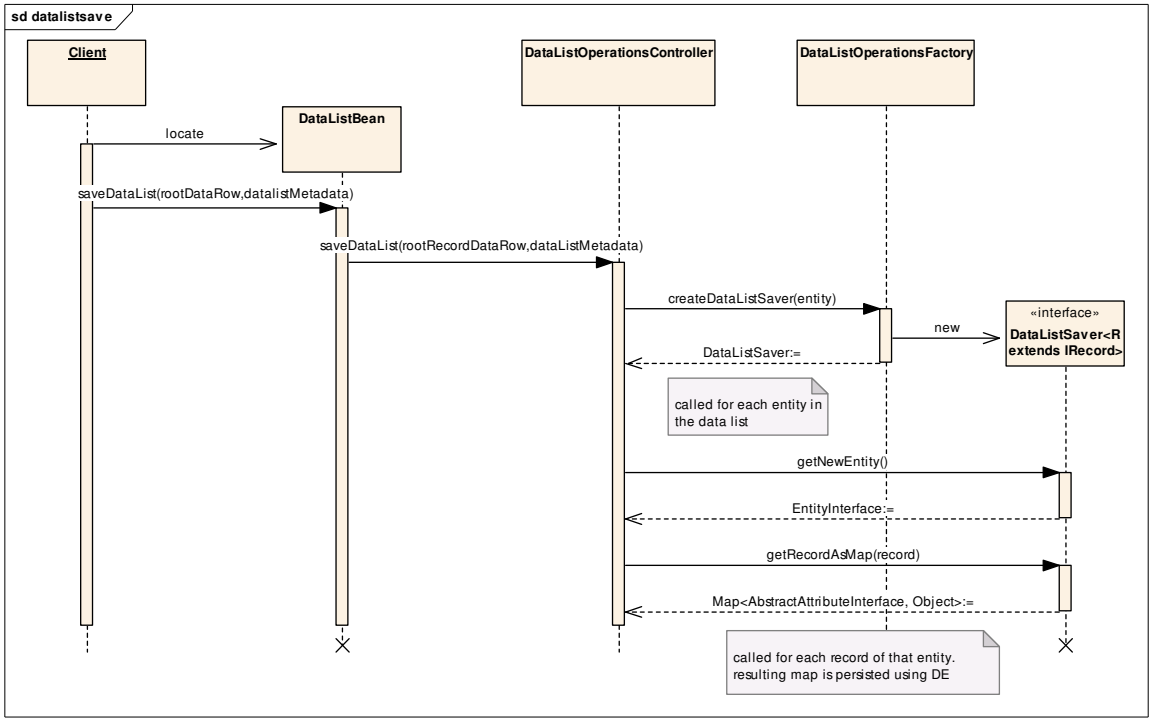

Figure 46 Sequence diagram for saving records of a data list

15 Experiment

15.1 Overview

User can create the experiment out of saved data-list. After creating user can perform various operations on it like visualizing data in the experiment using viewers, perform analysis or filtering the data etc.

15.2 Experiment Data Model

Following figure show the experiment data model.

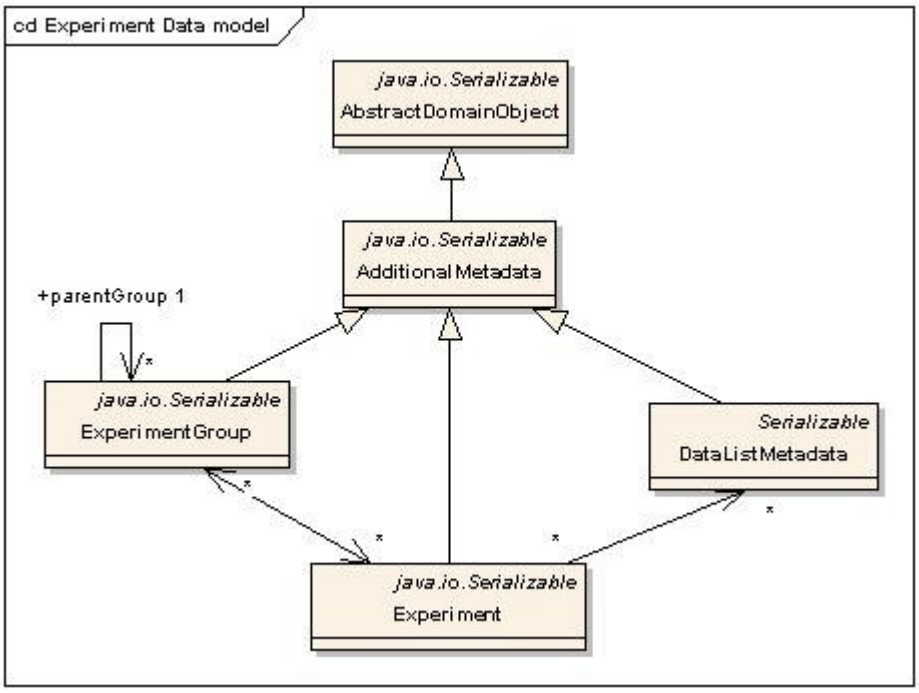

Figure 47 Experiment data model

- **AbstractDomainObject** is the base of all the domain objects in caB2B. It provides id and activity status fields required for all the domain objects
- **AdditionalMetadata** provides the additional information for the experiment and related objects. It includes name, description, created time and last updated time.
- **Experiment** contains the one or more **DataListMetadata.**
- **ExperimentGroup** is logical grouping of the experiments. It also has a parent group. This gives the folder (tree like) structure for experiment and experiment group.
- **DataListMetadata** is the object that represents the actual data saved. It contains the one or more entity ids that correspond to the DE entity ids created for the saved data.

15.3 Saving an Experiment

Following sequence diagram shows the flow of events while creating and saving a new experiment:

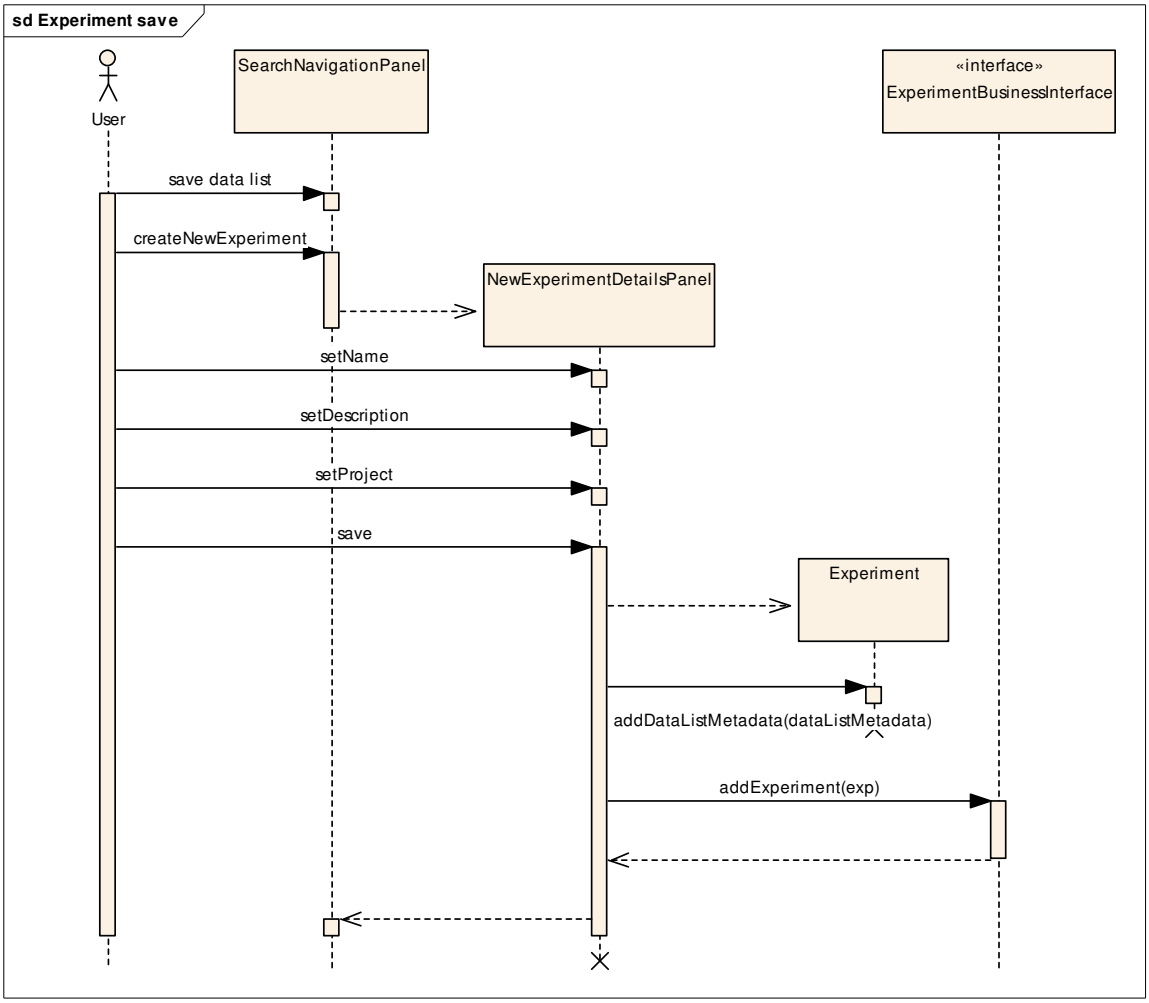

Figure 48 Flow of evens for saving experiment

15.4 Opening an Experiment

Following are the classes involved in displaying the experiment.

- **ExperimentPanel** is used to display details of all experiments. It is invoked when Experiment tab on GlobalNavigation panel is clicked. It contains ExperimentHierarchyPanel on left hand side and ExperimentDetailsPanel on right hand side.
- **ExperimentHierarchyPanel** is a panel to display experiments folder structure in the form of project and sub projects. On click of link in the tree details of the selected experiment or group are shown in ExperimentDetails panel.
- **ExperimentDetailsPanel** displays the information of the selected experiment group or experiment in a spreadsheet format. On the click of experiment name, ExperimentOpenPanel gets invoked**.**
- **ExperimentOpenPanel** is the main panel used to display the actual data in the selected experiment. It has ExperimentStackBox embedded in left hand side and ExperimentDataCategoryGridPanel embedded in right hand side.
- **ExperimentStackBox** is used to display data and the other tools that user can invoke on the data of the experiment. It contains panels to show the categories in the experiment. It also contains Filter panel, Visualization Panel and analytical services panel. On click of link in the

data category tree, the details of selected data category are shown in ExperimentDataCategoryGridPanel.

- **ExperimentDataCategoryGridPanel** is the base panel to display actual data in the experiment in the form of spreadsheet. It also acts as container for the dynamic tabs that gets added as user performs visualization and analytical tasks.

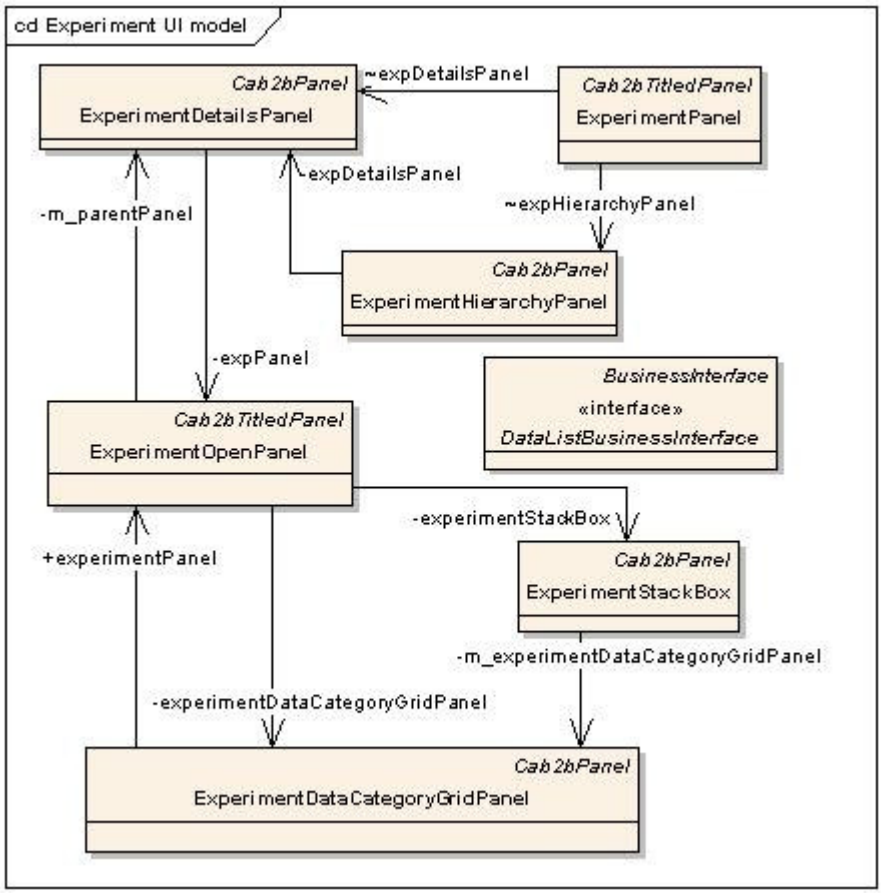

Figure 49 Experiment UI model

Following sequence diagram illustrates the flow of events while opening an experiment.

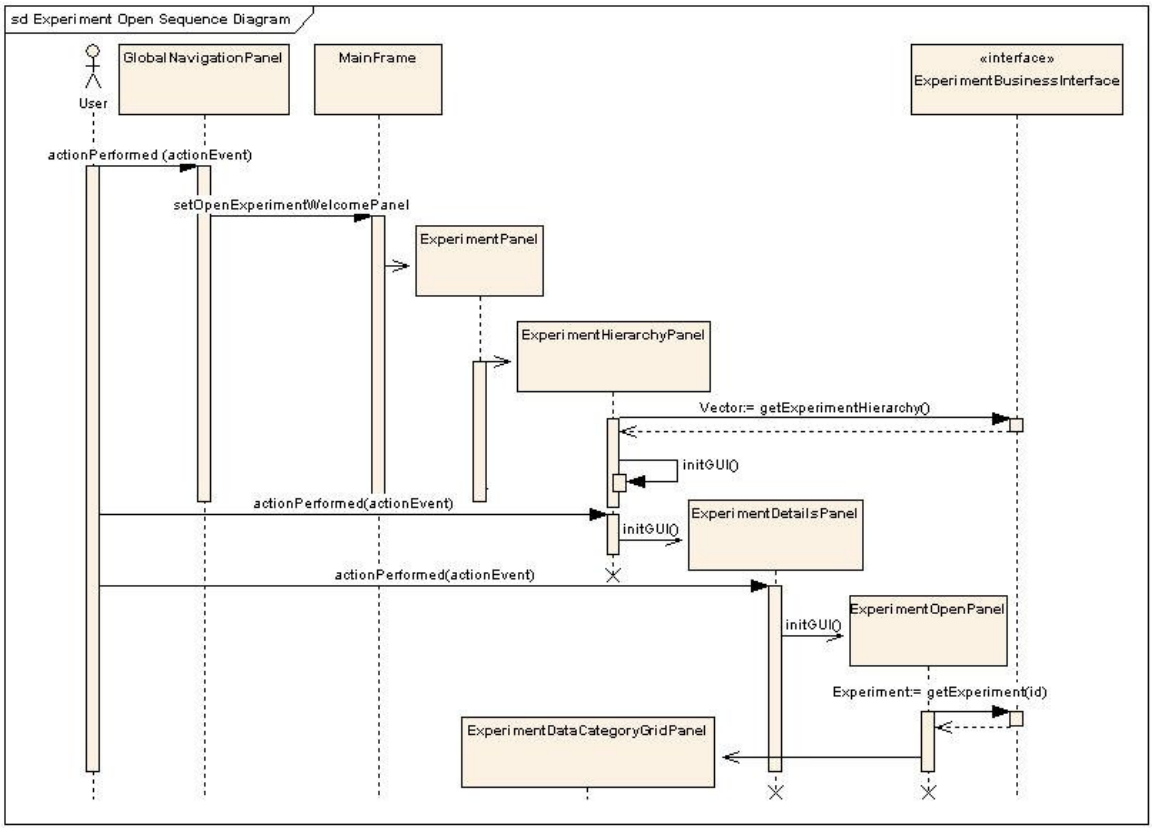

Figure 50 Flow of event for Open Experiment

15.5 Custom Data category

User can filter the data present in the experiment and save that sub set of the data as a custom data category. When user creates custom category, the current data present in the ExperimentDataCategoryGridPanel is taken and a new data list is created. This is distinguished with the other data list by setting its *isCustomDataCategory* flag to true. This data list is added is then saved along with its metadata and actual data. After this it is added into the current experiment and then UI is updated to reflect the change.

Following sequence diagram illustrates the flow of events while saving the custom data category.

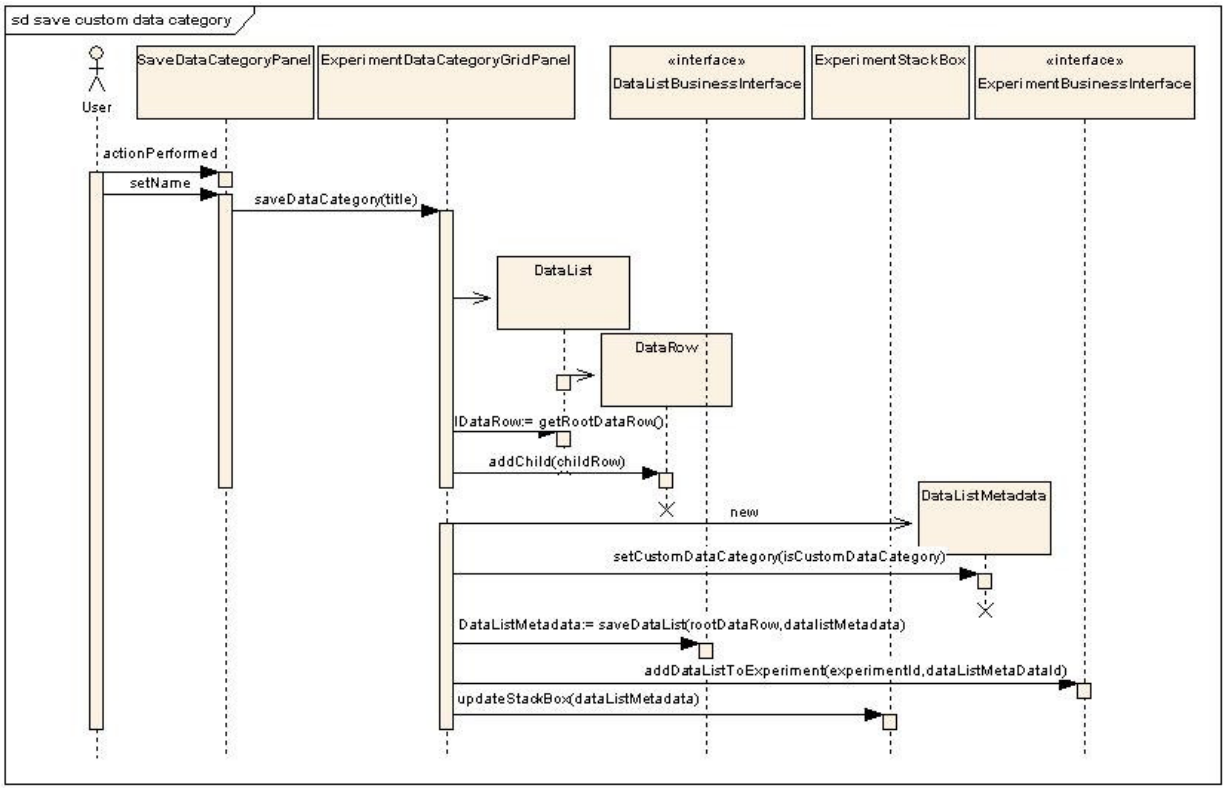

Figure 51 flow for saving the custom data category

16 Charting

16.1 Overview

The experiment data saved by the user after searching and saving the data list, can be scrutinize either by Analytical services or Visualization tools. Cab2b chart is one of the visualization options. It allows the user to see the various numerical data graphically by generating various charts out of it.

16.2 Classes Involved

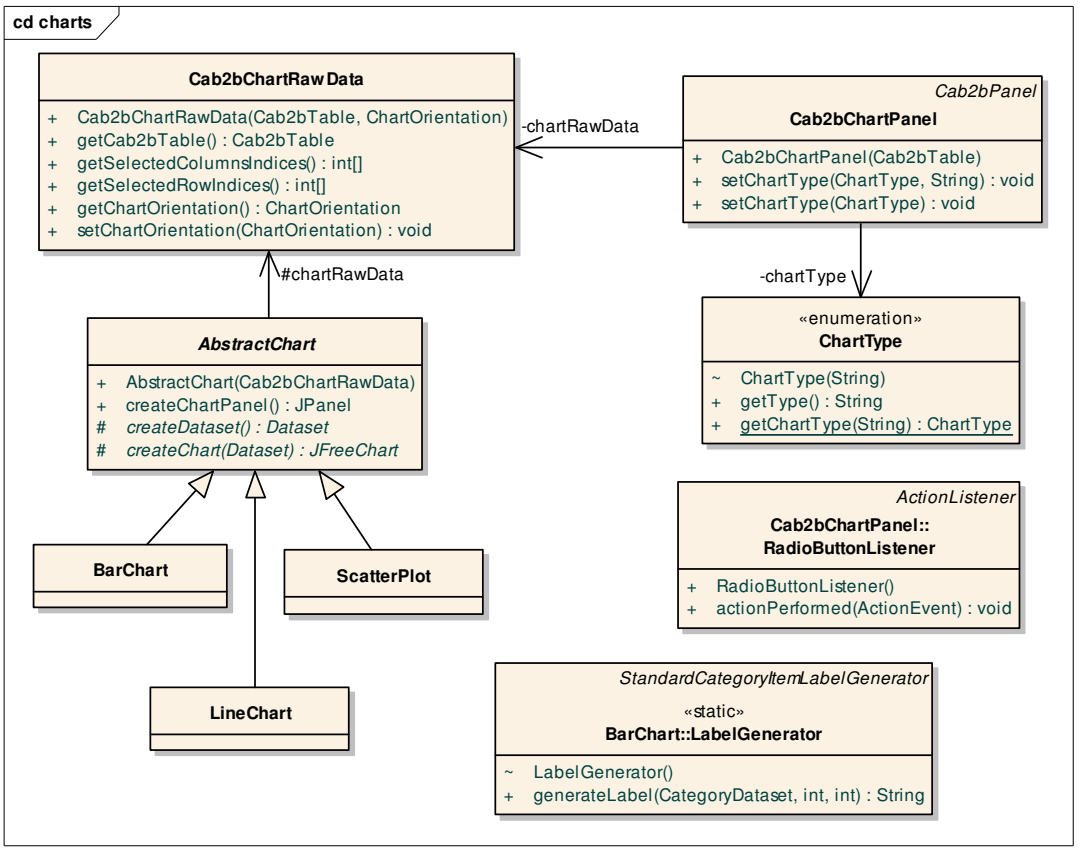

Figure 52 Classes Involved in Charting

Cab2bChartRawData stores the reference to the data table, the indices of the rows and columns selected in the data table, and the series of the chart (i.e. row wise or column wise) to be displayed.

ChartType is a wrapper around any of the following three types of charts that Cab2bChartPanel uses to decide which chart is to be rendered.

- BAR_CHART
- LINE_CHART
- SCATTER_PLOT

Cab2bChartPanel displays the requested chart. It also has the options to change the series of the chart. **Cab2bChartPanel**:**RadioButtonListener** acts on the selected option to change the series of the chart.

AbstractChart is the base class of all the chart classes. It holds the data to be rendered and provides a functionality that creates the chart panel.
BarChart is the chart class that renders the data to generate the bar chart.
BarChart:LabelGenerator is used by BarChart to generate the labels required in the chart.
LineChart is the chart class that renders the data to generate the line chart.
ScatterPlot is the chart class that renders the data to generate the scatter plot.

16.3 Sequence diagram

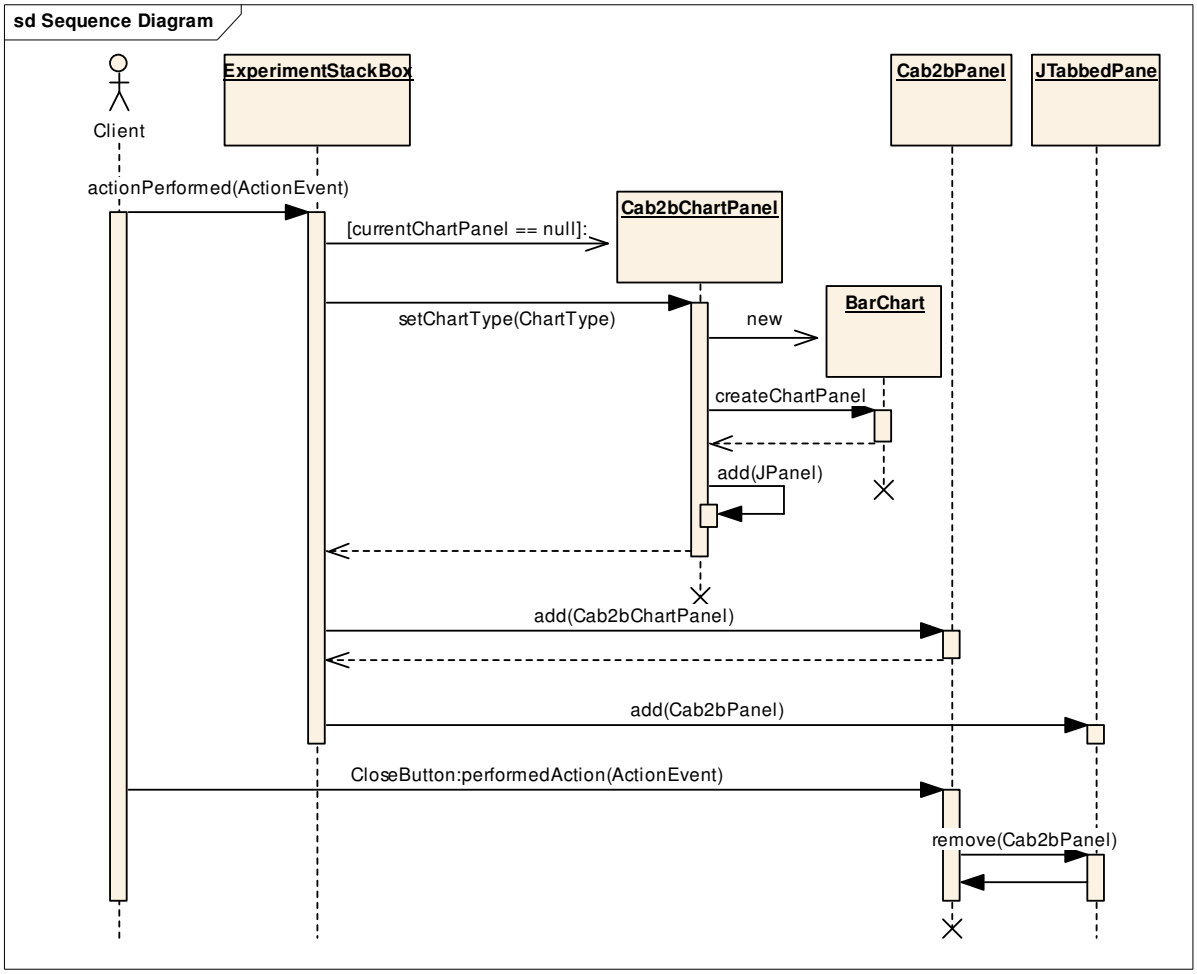

Figure 53 Flow of events happening during chart generation

Future functionalities

- Display large chart with scroll bars.
- To limit the legends of the chart to be displayed.

17 Filters

17.1 Overview

This component provides functionality to apply different types of filters on the table. JXTable provides basic functionality of filtering based on a pattern. It is extended to add few more types of filters. Based on the datatype of the specified column, following filters are supported

- Pattern filter
- Range Filter
- Enumerated Filter
- Boolean filter

Out of the above mentioned Pattern, Enumerated and Boolean filter are kind of pattern filer. Only Range filter is implemented in altogether different manner. Multiple filters can be applied on JXTable using FilterPipeline using method *setFilters(new FilterPipeline(filters));* Here filters is an array of filters where each element in this array is a filter with one condition on any column.

Filter component comprises of two basic components:

3. Popup to take the inputs
4. Actual filter to provide filtering action to the table.

17.2 Classes Involved

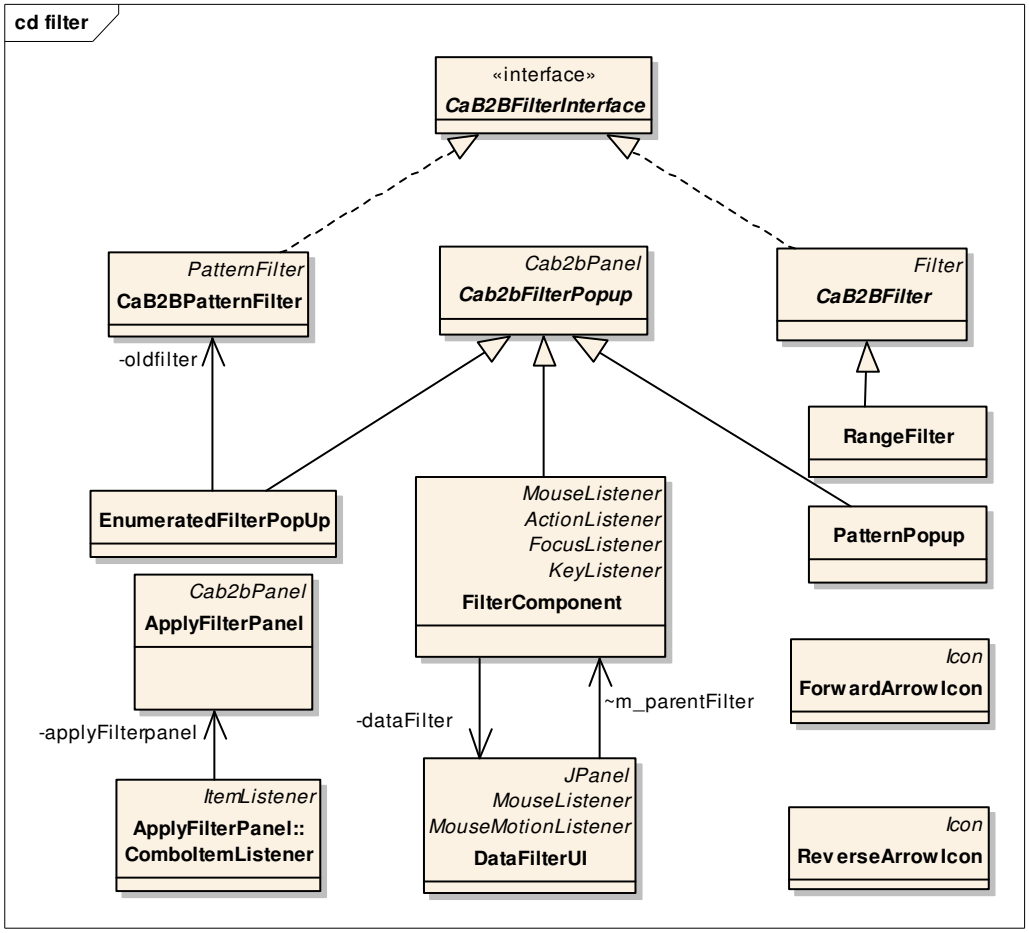

Figure 54 Classes Involved in Filtering data

ApplyFilterPanel creates a combo-box of all the headers of the columns in the present table and on click of its elements, calls an inner class ComboItemListener. This class also maintains a map filterMap of all the filters that are presently applied on the table.

ComboItemListener handles mouse click events. Method itemStateChanged() first finds the attribute datatype of the header clicked and accordingly instantiates subclass of Cab2bFilterPopup.

Cab2bFilterPopup is an abstract base class for all types of filter pop-ups. Its abstract method *okActionPerformed()* is called on the "OK" button click of filter pop-up. It returns CaB2BFilterInterface.

PatternPopup extends Cab2bFilterPopup. This class creates a pop-up asking user to enter desired search pattern. *okActionPerformed()* returns Cab2bPatternFilter.

FilterComponent class is a pop-up of range type. This class along with DataFilterUI, ForwardArrowIcon, ReverseArrowIcon forms user interface for range type filter pop-up. okActionPerformed() returns RangeFilter. This class primarily handles all the getters of min and max values for which the range is to be set.

DataFilterUI is responsible for actual UI implementation of the slider functionality. ForwardArrowIcon and ReverseArrowIcon generate icons which slide over the slider.

EnumeratedFilterPopUp is used for columns whose values can take limited/enumerated values. Its okActionPerformed() returns CaB2BPatternFilter. As enumerated filter is nothing but pattern filter with multiple patterns, this class, while performing OK button action, connects all the selected patterns to create one single pattern. This pattern then is used to create and return an instance of CaB2BPatternFilter.

CaB2BFilterInterface is common interface for all the custom filters. Its copy() method creates copy of CaB2BFilterInterface and returns. This method is responsible for creating a copy of a filter. It is called while applying filters over the table. Values in filterMap from ApplyFilterPanel are never used directly to create an array of filters to be applied on the table. A copy of each filter from filterMap is created and that copy is added to current array of filters. This is done because when a filter is edited, we need to first check prior values of the same. Thus reference copy is necessary.

CaB2BPatternFilter extends java class PatternFilter. It has additional toString() method to create a text of currently applied filter. It is used to display currently applied filters to user.
RangeFilter is a modified a PatternFilter accommodating range inputs and filtering.

17.3 Sequence diagram

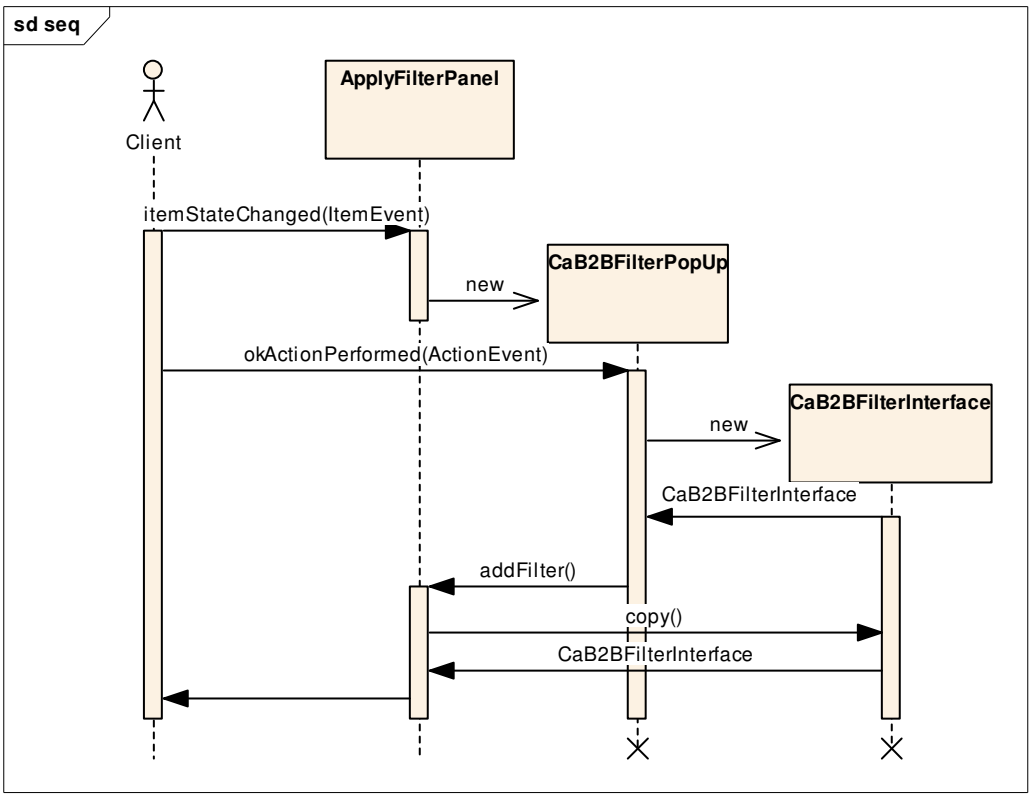

Figure 55 Flow of events happening when user applies a filter

18 Analytical Services Invoker

18.1 Overview

Analytical services are the services which transforms data from one form to another by applying some algorithm on it. When user is viewing records of a particular entity, analytical services applicable for that entity are shown in left-hand-side stack box.

18.2 Entity to Analytical Service Mapping XML

Finding analytical services for an entity is a metadata driven process. A file *EntityToAnalyticalServiceMapping.xml* is used to find that. Figure below shows a sample of that configuration file.

```
<entity_service_mapping>
  <service name="CMS"
           URL="http://node255.broad.mit.edu:6060/wsrf/services/cagrid/ComparativeMarkerSelMAGESvc">
      <method name="invoke"
              serviceDetailClass="edu.wustl.cab2b.common.analyticalservice.CMSServiceDetails"
              serviceInvokerClass="edu.wustl.cab2b.server.analyticalservice.CMSServiceInvoker"
      />
  </service>

  <service name="Service1" URL="">
      <method name="method1" serviceDetailClass="TempDetail1" serviceInvokerClass="TempInvoker"/>
      <method name="method2" serviceDetailClass="TempDetail2" serviceInvokerClass="TempInvoker"/>
  </service>

  <service name="Service2" URL="">
      <method name="method1" serviceDetailClass="" serviceInvokerClass=""/>
  </service>

  <service name="Service3" URL="">
      <method name="method1" serviceDetailClass="" serviceInvokerClass=""/>
  </service>

  <entity name="gov.nih.nci.mageom.domain.BioAssay.BioAssay" serviceName="CMS"/>
  <entity name="Entity1" serviceName="Service1"/>
  <entity name="Entity2" serviceName="Service2"/>
  <entity name="Entity2" serviceName="Service3"/>
  <entity name="Entity3" serviceName="Service3"/>
</entity_service_mapping>
```

Figure 56 Sample EntityToAnalyticalServiceMapping.xml

- **<entity>:** This file has <entity> tag which specifies which gives mapping between entity and its one applicable service. There can be multiple services applicable for an entity. For this there will be those many <entity> tags with different service names.
- **<service>:** This file has one <service> tag for one service. It has a unique name of the service which is shown to the user and the URL pointing to the running instance of that service.
- **<method>** tag in the service states which method of that service is to be invoked. Attribute **serviceDetailsClass** gives the class which holds details of the service. The class mentioned here must implement **ServiceDetailsInterface.** There is a method on this interface *getRequiredEntities ()* which returns list of entities. One of them will be the one for which user is currently viewing the data. For other entities a dynamic UI is generated to specify values for its attributes. Attribute **serviceInvokerClass** specifies which class to be used to invoke the service. The class mentioned here must implement **ServiceInvokerInterface.**

18.3 Classes involved

Diagram below shows the classes involved at the backend. It also shows implementation done for comparative marker selection analytical service.

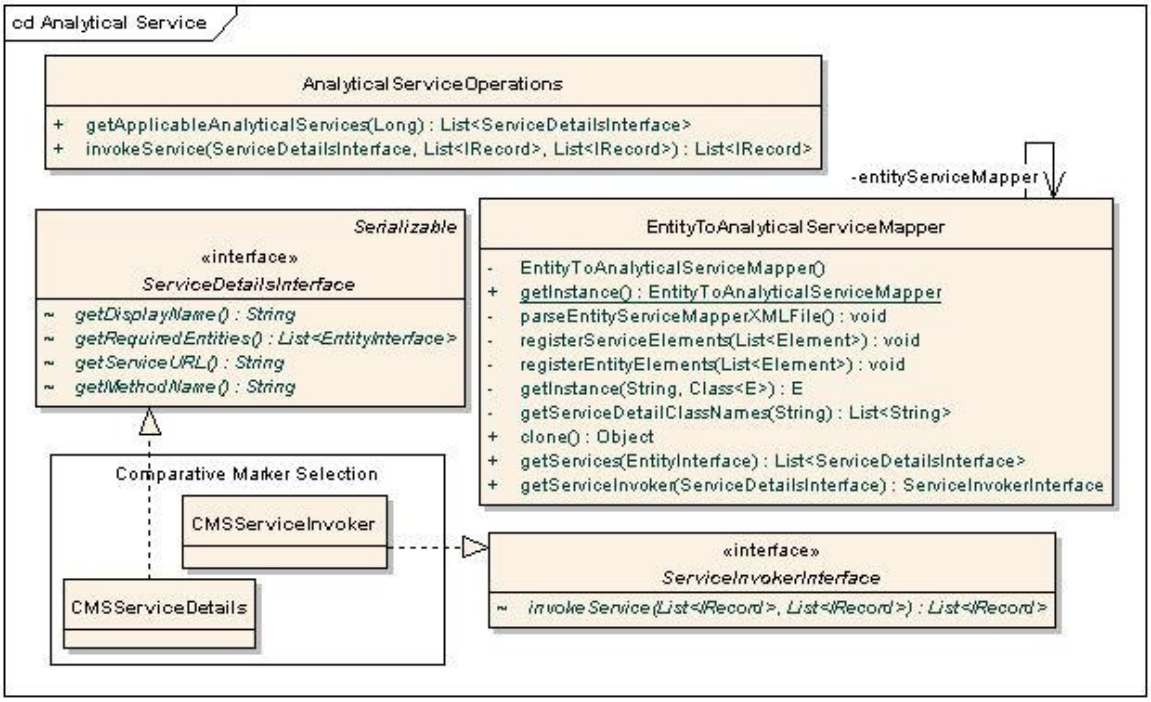

Figure 57 Classes involved in getting and invoking analytical services

- **ServiceDetailsInterface:** It defines the methods needed to describe any analytical service like its name, required entities, URL pointing to service instance. All the classes mentioned as value of attribute **serviceDetailsClass** in above XML file must implement this interface
- **ServiceInvokerInterface:** It defines the method to invoke an analytical service. All the classes mentioned as value of attribute **serviceInvokerClass** in above XML file must implement this interface
- **EntityToAnalyticalServiceMapper:** This is a singleton class which parses the *EntityToAnalyticalServiceMapping.xml* file and stores the mapping information into an internal map. This class provides the methods to get the service interface and the service invoker interface.
- **AnalyticalServiceOperations:** This class has a method to get applicable analytical services which returns list of **ServiceDetailsInterface** for a given entity. It also has method invoke () to call the service with passed data.
- **CMSServiceDetails** and **CMSServiceInvoker** are the real extensions implemented to invoke comparative marker selection analytical service.

19 Appendix

19.1 Dynamic Extension and MDR

19.1.1 Overview

One of the most important components of the DE project is its metadata repository. MDR can contain metadata about dynamic extensions or static UML models. Each DE is also a UML model. The MDR is very important component not just for DE, but also for applications like caB2B and caTissue Suite. The basic backbone of MDR is as shown in Figure 1 Metadata Repository backbone.

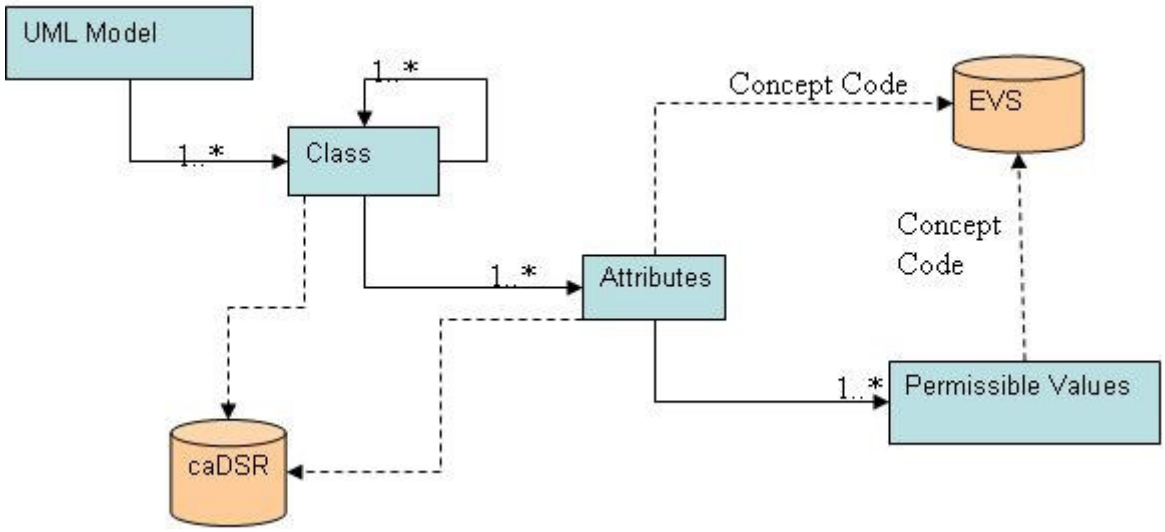

Figure 58 Metadata Repository backbone

MDR contains the following metadata for a domain model:

- Classes
- Attributes
- Data type
- Concept codes
- Description
- Permissible values

In case the domain model is created using the dynamic extensions user interface, the MDR will contain the UI display properties and the database mapping information for each attribute. The metadata for the user interface contains:

- Type of UI Control
- Properties like height, width, password like string and so forth
- Mandatory or optional attribute

Table to which the entity maps and column to which the attribute maps

19.1.2 UML Metadata

This contains all the information present in the UML model like class, attributes, and associations including the permissible values. Following diagram shows the classes involved in entity creation along with the relationships involved in these classes.

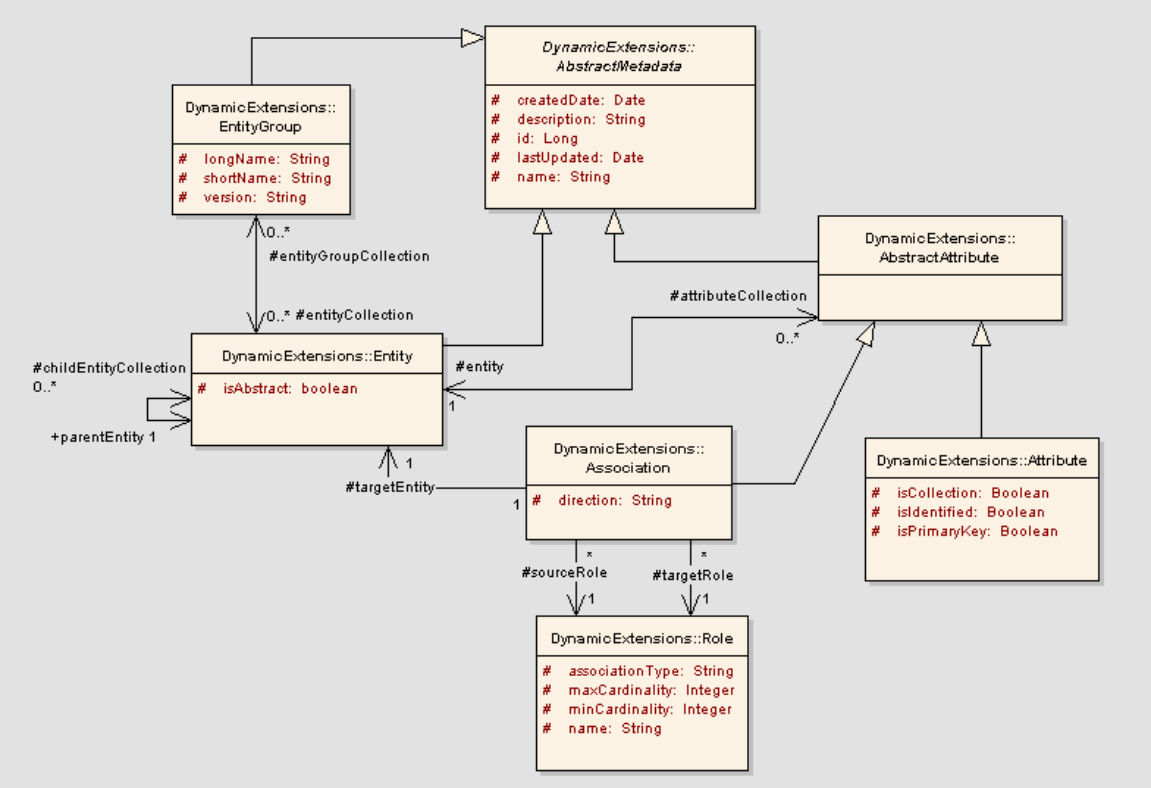

Figure 59 Dynamic extension basic metadata

AbstractMetadata: This is an abstract base class from which the backbone metadata objects are derived. This class contains generic attributes which are part of all objects (like create date, last updated and so forth).
EntityGroup: An entity group is a logical collection of entities. For example, all classes of an application are loaded under one entity group. It contains multiple entities.
Entity: This class represents an UML class. An entity is associated to itself to specify its parent entity. An entity can have zero or one parent entity. An entity can also have zero or more children entities.

AbstractAttribute: An entity can either have zero or more primitive attributes, or have zero or more associated classes. This is represented by the AbstractAttribute class. It is the base class for Association and Attribute classes.
Attribute: The class represents a primitive attribute. For example, name is an attribute of the user entity. Attribute can be of following types:

- String attribute
- Double attribute
- Short attribute
- Long attribute
- Boolean Attribute
- Date attribute
- ByteArray (for BLOB/CLOB)

Following diagram shows how attribute type is defined or changed in attribute.

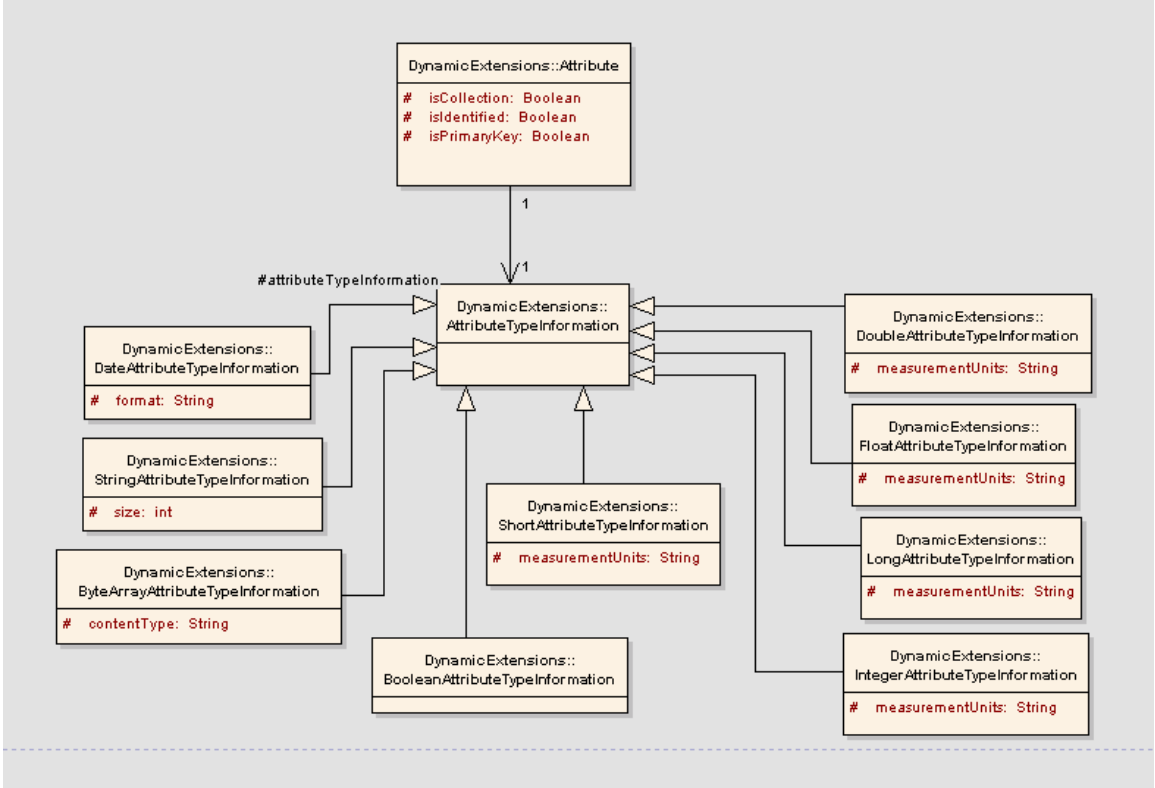

Figure 60 Attribute Type Metadata

Attribute class is associated with the class "AttributeTypeInformation" that specifies the type of the attribute.
AttributeTypeInformation: This class represents the type of the attribute. Attribute type can be any of the above mentioned types. This class is an abstract class which is extended by all the specific primitive attribute types like DoubleAttributeTypeInformation or StringAttributeTypeInformation.

Role: This class describes an association's cardinality and the association type. The class has the following attributes
associationType: This could be two types of association: containment or linking.
Containment association type is one of Person and Address where the Person entity will contain Address entity within it. The Address object does not exist on its own. Linking association type is one of User and Study. Here, both the objects can be created independently. The user can be part of multiple studies and a study can contain multiple users.
maxCardinality: Maximum cardinality of association (for example, 1 or many)
minCardinality: Maximum cardinality of association (for example, 0, 1 or many)
name: The role name of the association.

Association: This class represents the associations that an entity can have with other entities. E.g. a User entity is associated with Institute entity.
sourceRole: This represents the role of the association from the source context.
targetRole: This represents the role of the association from the target context.

19.1.3 Inheritance Metadata support

One of the main aspects of any application is the inheritance between its entities. So when any object model is loaded into DE database, this hierarchy of objects should be preserved.

This section explains how inheritance is preserved in DE using the required metadata objects of DE. Following diagram explains the required objects and relationships for inheritance.

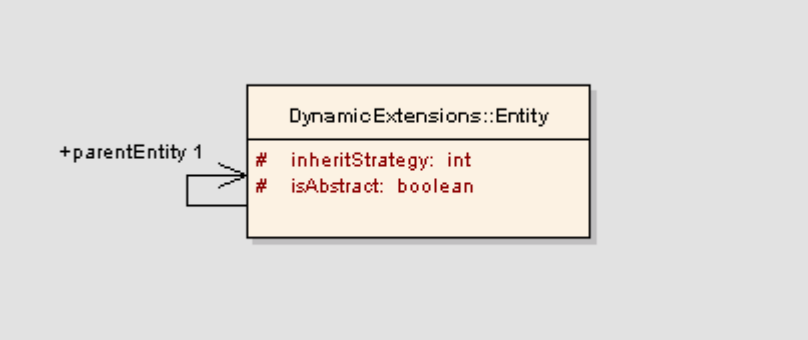

Figure 61 Inheritance Metadata

Entity: Entity object represents the java class in any object model. So to maintain the hierarchy of classes, following attributes and associations are maintained.

isAbstract: This flag maintains whether the entity is abstract or not.
inheritStrategy: This attribute stores the Hibernate's strategy to store the actual data in the actual database. Allowed values for this attribute are:

1. Joined subclass
2. Subclass
3. Table per concrete class.

19.1.4 Attribute data elements and default values

An attribute can have values that are derived from some fixed source or some user defined set of allowable values. For example, gender attribute can have only fixed values like male and female. Additionally, the attribute can have one of them as a default value. This information is saved in following way. The diagram shows the way in which the allowable and default values are stored in DE

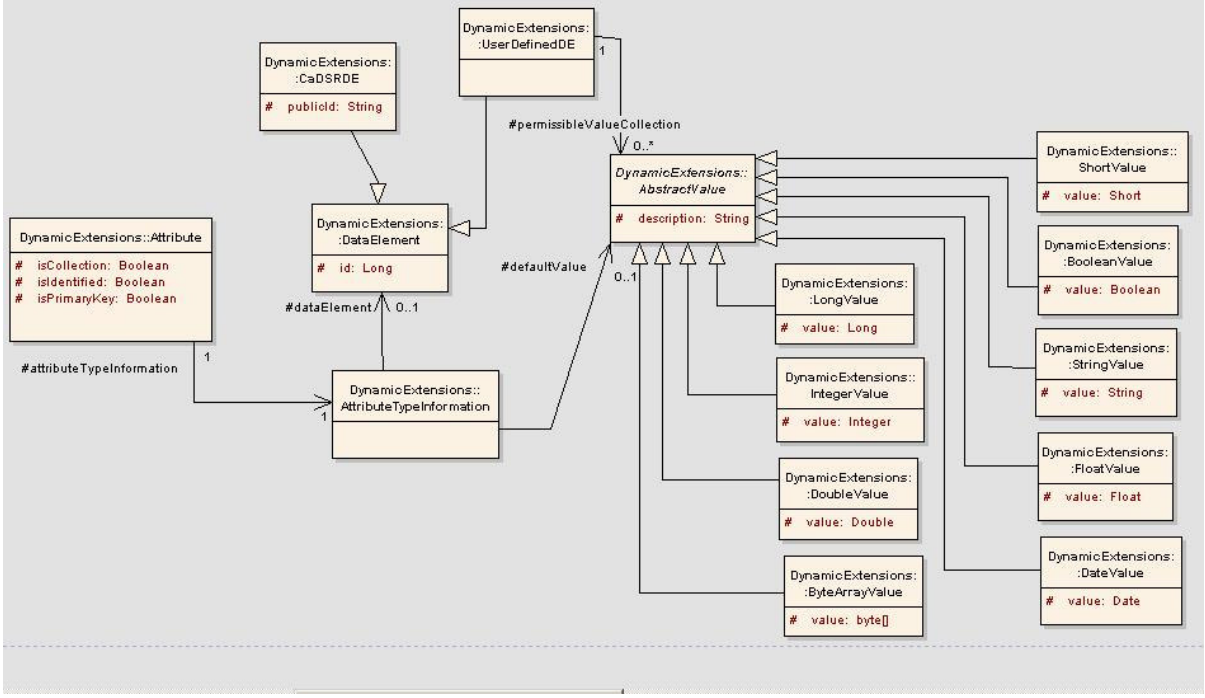

Figure 62 Attribute Data Elements

caDSRDE holds all the common information for all the types of data elements. Some of the associations of this class are:

- *AttributeTypeInformation*: Source of the allowable values is specific to the attribute type. So to represent this information correctly, AttributeTypeInformation class is associated with the DataElement so that it represents the type of source for the attribute.
- *AbstractValue*: This class represents a value, an attribute can have. This value can be used as a default value or as one of the allowable values. The class acts as a base class for the entire attribute type specific value